



**GHULAM ISHAQ KHAN INSTITUTE OF
ENGINEERING SCIENCES AND
TECHNOLOGY**

FACULTY OF COMPUTER SCIENCE AND ENGINEERING

**CS342 – NUMERICAL ANALYSIS PROJECT
REPORT**

SUBMITTED TO: SIR AAMIR SHEHZAD

DATED: 5TH MAY, 2025

GROUP MEMBERS:

JUNAID SALEEM--2022243

ABDUL MUEED KHAN--2022013

MUAZZAM SHAH--2022312

DUA-E-ZAHRA NAQVI--2022151

NEHA ABDUL RAHIM--2022481

Work Distribution

Junaid Saleem—2022243

- Newton-Raphson Method
- Cubic Spline Interpolation
- Parametric Curves
- Numerical Integration
- Report Structuring and Formatting

Abdul Mueed Khan—2022013

- Bisection Method
- Secant Method
- Lagrange Interpolation
- Richardson Extrapolation

Muazzam Shah—2022312

- Fixed Point Iteration
- Hermite Interpolation
- Numerical Differentiation
- Romberg Integration

Dua-e-Zahra Naqvi—2022151

- Matrix Factorization
 - LU Factorization
 - Doolittle Factorization
 - Crout Factorization
 - Cholesky Factorization

Neha Abdul Rahim—2022481

- Gaussian Elimination and Variants
- Pivoting Strategies
 - Partial Pivoting
 - Scaled Partial Pivoting
 - Complete Pivoting

Table of Contents

1	Bisection Method	6
1.1	Method Overview	6
1.2	Explanation of the Code.....	6
1.3	Tables of Results	7
1.4	Graphs	9
1.5	Interpretation of Results.....	10
2	Fixed Point Iteration	11
2.1	Method Overview	11
2.2	Explanation of Code	11
2.3	Table of Results	12
2.4	Graphs	14
2.5	Interpretation of Results.....	15
3	Newton-Raphson Method	16
3.1	Method Overview	16
3.2	Explanation of the Code.....	16
3.3	Table of Results	17
3.4	Graphs	18
3.5	Interpretation of Results.....	19
4	Secant Method	19
4.1	Method Overview	19
4.2	Explanation of the Code.....	20
4.3	Table of Results	20
4.4	Graphs	21
4.5	Interpretation of Results.....	22
5	Lagrange Interpolation.....	23
5.1	Method Overview	23
5.2	Explanation of the Code.....	23
5.3	Table of Results	24
5.4	Graphs	26
5.5	Interpretation of Results.....	27
6	Hermite Interpolation.....	28
6.1	Method Overview	28
6.2	Explanation of Code	29
6.3	Table of Results	29

6.4	Graphs	32
6.5	Interpretation of Results.....	33
7	Cubic Spline Interpolation	34
7.1	Method Overview	34
7.2	Explanation of Code	34
7.3	Table of Results	35
7.4	Graphs.....	38
7.5	Interpretation of Results.....	39
8	Parametric Curves.....	40
8.1	Method Overview	40
8.2	Explanation of Code	40
8.3	Graph and Data	41
9	Numerical Differentiation.....	45
9.1	Method Overview	45
9.2	Explanation of Code	45
9.3	Results and Graphs	46
9.4	Interpretation of Results.....	50
9.5	Conclusion	50
10	Richardson Extrapolation.....	51
10.1	Method Overview	51
10.2	Explanation of Code	51
10.3	Table and Graphs	52
10.4	Interpretation of Results.....	55
11	Numerical Integration	56
11.1	Method Overview	56
11.2	Explanation of Code	56
11.3	Results and Graphs	57
11.4	Interpretation of Results.....	60
12	Romberg Integration	62
12.1	Method Overview	62
12.2	Explanation of Code	62
12.3	Results and Graphs:	63
12.4	Interpretation of Results.....	67
13	Gaussian Elimination and Variants.....	68
13.1	Standard Gaussian Elimination.....	68

13.1.1	Forward Elimination	68
13.1.2	Back Substitution:	69
14	Pivoting Strategies	69
14.1	Gaussian Elimination with Partial Pivoting:	69
14.2	Gaussian Elimination with Complete Pivoting:	71
14.3	Gaussian Elimination with Scaled Partial Pivoting	73
15	Matrix Factorization.....	74
15.1	LU Decomposition.....	74
15.2	DoLittle Decomposition.....	76
15.3	Crout LU Decomposition.....	77
15.4	Cholesky LU Decomposition.....	79

1 Bisection Method

1.1 Method Overview

- Purpose: To find a root of a continuous function $f(x)$ in an interval $[a,b]$ where $f(a) \cdot f(b) < 0$.
- Based on the Intermediate Value Theorem.
- Steps:
 - Choose an interval $[a, b]$ with opposite signs of $f(a)$ and $f(b)$.
 - Calculate midpoint $c = \frac{a+b}{2}$
 - Evaluate $f(c)$.
 - If $f(c)=0$, root is found.
 - If $f(a) \cdot f(c) < 0$, set $b = c$; else set $a=c$.
 - Repeat until desired precision is achieved.
- Convergence: Guaranteed if the initial condition is met; linear and slow.
- Stopping Criteria: Maximum iterations, interval width below tolerance, or $|f(c)|$ below threshold.
- Advantages: Simple, always converges if conditions are met.
- Disadvantages: Slow, requires sign change over interval.

1.2 Explanation of the Code

The implementation consists of three main functions:

bisectionMethod:

- Takes function, interval bounds $[a, b]$, tolerance, and max iterations
- Returns root approximation, iteration count, and iteration data
- Checks for sign change at interval endpoints
- Updates interval based on function value at midpoint

printResults:

- Shows current interval bounds $[a, b]$ and their function values
- Displays midpoint (x) and its function value
- Calculates error between consecutive approximations
- Formats output in a clear tabular form

plotResults:

- Error vs Iterations plot showing convergence
- Function curve plot with root location

1.3 Tables of Results

Stopping Tolerance: 10^{-6}

$$f(x) = x^3 - 4x + 1$$

Interval: [0,1]

Iteration	a	f(a)	b	f(b)	x	f(x)	Error
1	0.000000	1.000000	1.000000	-2.000000	0.500000	-0.875000	inf
2	0.000000	1.000000	0.500000	-0.875000	0.250000	0.015625	0.250000
3	0.250000	0.015625	0.500000	-0.875000	0.375000	-0.447266	0.125000
4	0.250000	0.015625	0.375000	-0.447266	0.312500	-0.219482	0.062500
5	0.250000	0.015625	0.312500	-0.219482	0.281250	-0.102753	0.031250
6	0.250000	0.015625	0.281250	-0.102753	0.265625	-0.043758	0.015625
7	0.250000	0.015625	0.265625	-0.043758	0.257812	-0.014114	0.007812
8	0.250000	0.015625	0.257812	-0.014114	0.253906	0.000744	0.003906
9	0.253906	0.000744	0.257812	-0.014114	0.255859	-0.006688	0.001953
10	0.253906	0.000744	0.255859	-0.006688	0.254883	-0.002973	0.000977
11	0.253906	0.000744	0.254883	-0.002973	0.254395	-0.001115	0.000488
12	0.253906	0.000744	0.254395	-0.001115	0.254150	-0.000185	0.000244
13	0.253906	0.000744	0.254150	-0.000185	0.254028	0.000279	0.000122
14	0.254028	0.000279	0.254150	-0.000185	0.254089	0.000047	0.000061
15	0.254089	0.000047	0.254150	-0.000185	0.254120	-0.000069	0.000031
16	0.254089	0.000047	0.254120	-0.000069	0.254105	-0.000011	0.000015
17	0.254089	0.000047	0.254105	-0.000011	0.254097	0.000018	0.000008
18	0.254097	0.000018	0.254105	-0.000011	0.254101	0.000003	0.000004
19	0.254101	0.000003	0.254105	-0.000011	0.254103	-0.000004	0.000002
20	0.254101	0.000003	0.254103	-0.000004	0.254102	-0.000000	0.000001

$$f(x) = \sqrt{x} - 2$$

Interval: [0,10]

Iteration	a	f(a)	b	f(b)	x	f(x)	Error
1	0.000000	-2.000000	10.000000	1.162278	5.000000	0.236068	inf
2	0.000000	-2.000000	5.000000	0.236068	2.500000	-0.418861	2.500000
3	2.500000	-0.418861	5.000000	0.236068	3.750000	-0.063508	1.250000
4	3.750000	-0.063508	5.000000	0.236068	4.375000	0.091650	0.625000
5	3.750000	-0.063508	4.375000	0.091650	4.062500	0.015564	0.312500
6	3.750000	-0.063508	4.062500	0.015564	3.906250	-0.023576	0.156250
7	3.906250	-0.023576	4.062500	0.015564	3.984375	-0.003910	0.078125

Iteration	a	f(a)	b	f(b)	x	f(x)	Error
8	3.984375	-0.003910	4.062500	0.015564	4.023438	0.005851	0.039062
9	3.984375	-0.003910	4.023438	0.005851	4.003906	0.000976	0.019531
10	3.984375	-0.003910	4.003906	0.000976	3.994141	-0.001465	0.009766
11	3.994141	-0.001465	4.003906	0.000976	3.999023	-0.000244	0.004883
12	3.999023	-0.000244	4.003906	0.000976	4.001465	0.000366	0.002441
13	3.999023	-0.000244	4.001465	0.000366	4.000244	0.000061	0.001221
14	3.999023	-0.000244	4.000244	0.000061	3.999634	-0.000092	0.000610
15	3.999634	-0.000092	4.000244	0.000061	3.999939	-0.000015	0.000305
16	3.999939	-0.000015	4.000244	0.000061	4.000092	0.000023	0.000153
17	3.999939	-0.000015	4.000092	0.000023	4.000015	0.000004	0.000076
18	3.999939	-0.000015	4.000015	0.000004	3.999977	-0.000006	0.000038
19	3.999977	-0.000006	4.000015	0.000004	3.999996	-0.000001	0.000019
20	3.999996	-0.000001	4.000015	0.000004	4.000006	0.000001	0.000010
21	3.999996	-0.000001	4.000006	0.000001	4.000001	0.000000	0.000005
22	3.999996	-0.000001	4.000001	0.000000	3.999999	-0.000000	0.000002
23	3.999999	-0.000000	4.000001	0.000000	4.000000	-0.000000	0.000001
24	4.000000	-0.000000	4.000001	0.000000	4.000000	0.000000	0.000001

$$f(x) = \cos(x) - x$$

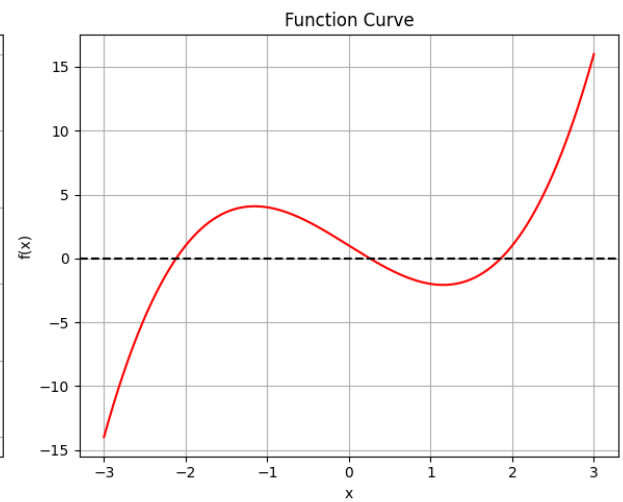
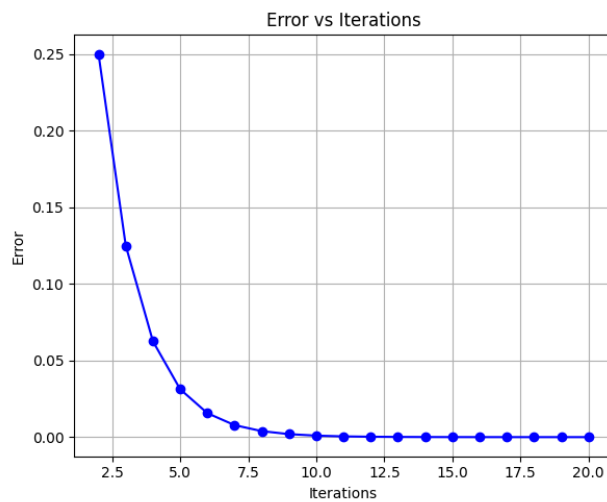
Interval: [0,1]

Iteration	a	f(a)	b	f(b)	x	f(x)	Error
1	0.000000	1.000000	1.000000	-0.459698	0.500000	0.377583	inf
2	0.500000	0.377583	1.000000	-0.459698	0.750000	-0.018311	0.250000
3	0.500000	0.377583	0.750000	-0.018311	0.625000	0.185963	0.125000
4	0.625000	0.185963	0.750000	-0.018311	0.687500	0.085335	0.062500
5	0.687500	0.085335	0.750000	-0.018311	0.718750	0.033879	0.031250
6	0.718750	0.033879	0.750000	-0.018311	0.734375	0.007875	0.015625
7	0.734375	0.007875	0.750000	-0.018311	0.742188	-0.005196	0.007812
8	0.734375	0.007875	0.742188	-0.005196	0.738281	0.001345	0.003906
9	0.738281	0.001345	0.742188	-0.005196	0.740234	-0.001924	0.001953
10	0.738281	0.001345	0.740234	-0.001924	0.739258	-0.000289	0.000977
11	0.738281	0.001345	0.739258	-0.000289	0.738770	0.000528	0.000488
12	0.738770	0.000528	0.739258	-0.000289	0.739014	0.000120	0.000244
13	0.739014	0.000120	0.739258	-0.000289	0.739136	-0.000085	0.000122
14	0.739014	0.000120	0.739136	-0.000085	0.739075	0.000017	0.000061
15	0.739075	0.000017	0.739136	-0.000085	0.739105	-0.000034	0.000031

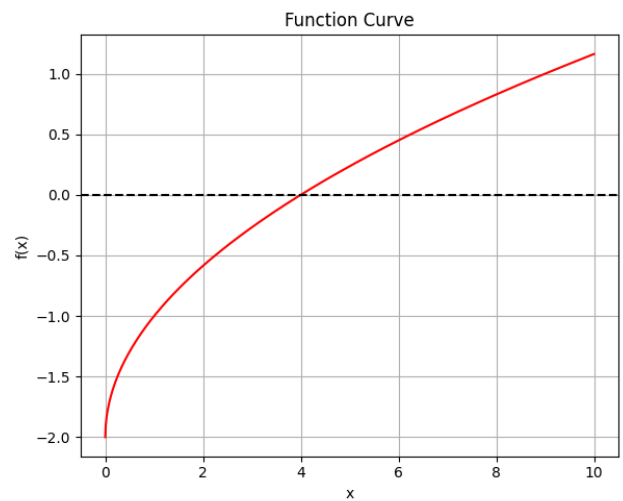
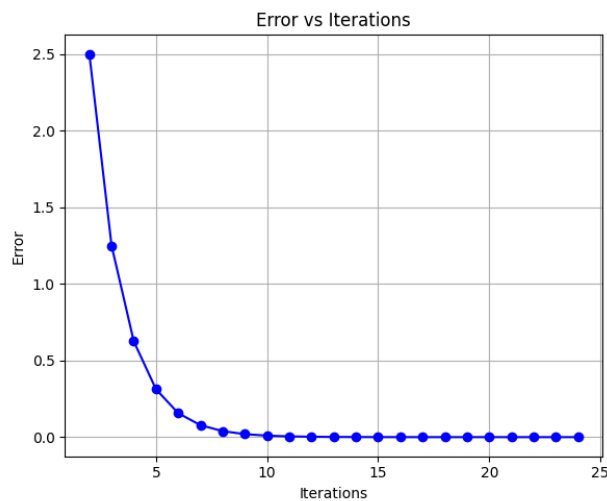
Iteration	a	f(a)	b	f(b)	x	f(x)	Error
16	0.739075	0.000017	0.739105	-0.000034	0.739090	-0.000008	0.000015
17	0.739075	0.000017	0.739090	-0.000008	0.739082	0.000005	0.000008
18	0.739082	0.000005	0.739090	-0.000008	0.739086	-0.000002	0.000004
19	0.739082	0.000005	0.739086	-0.000002	0.739084	0.000001	0.000002
20	0.739084	0.000001	0.739086	-0.000002	0.739085	-0.000000	0.000001

1.4 Graphs

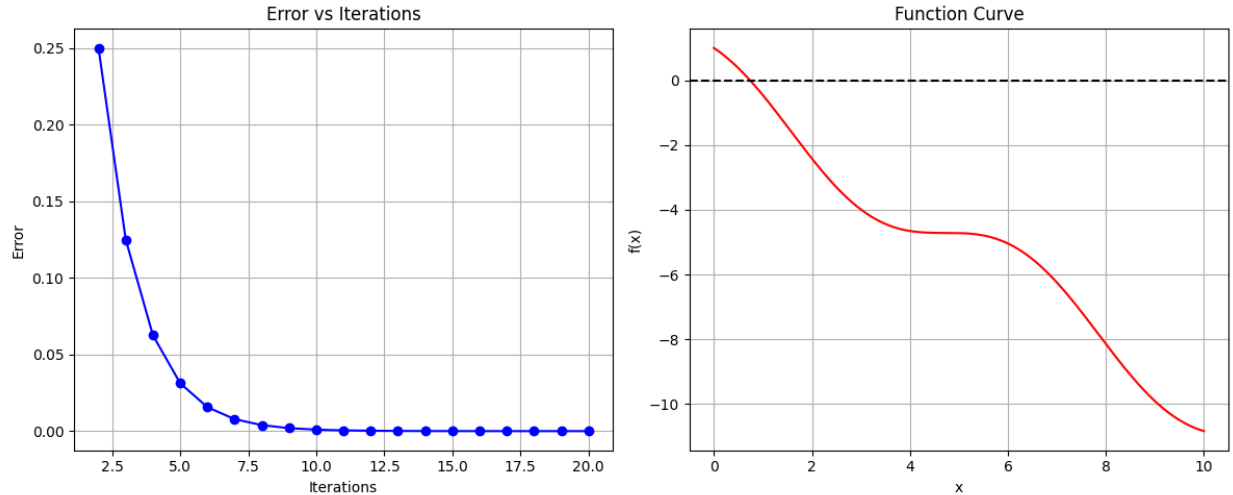
$$f(x) = x^3 - 4x + 1$$



$$f(x) = \sqrt{x} - 2$$



$$f(x) = \cos(x) - x$$



1.5 Interpretation of Results

The bisection method was applied to three functions, each with a known sign change in the given interval. In all cases, the method successfully converged to a root within the specified tolerance of 10^{-6} .

1.5.1 Function 1: $(x) = x^3 - 4x + 1$ on $[0, 1]$

- **Root found:** ≈ 0.254102
- **Iterations:** 20
- **Remarks:** Function changes sign over $[0,1]$; convergence was steady and accurate.

1.5.2 Function 2: $f(x)=\sqrt{x} - 2$ on $[0, 10]$

- **Root found:** ≈ 4.000000
- **Iterations:** 24
- **Remarks:** Root at $x=4x = 4$ was accurately detected; error halved each step.

1.5.3 Function 3: $f(x)=\cos(x)-x$ on $[0, 1]$

- **Root found:** ≈ 0.739085
- **Iterations:** 20
- **Remarks:** Classic fixed-point problem; bisection method converged reliably.

1.5.4 Conclusion

The method showed consistent convergence in all cases, with clear reduction in error and root approximation through midpoint updates. Tabulated results confirm the method's precision and stability.

2 Fixed Point Iteration

2.1 Method Overview

- Purpose: To find a root of $f(x) = 0$ by rewriting it as $x=g(x)$ and iterating.
- Based on the idea that a solution to $x=g(x)$ is a fixed point of the function $g(x)$.
- Steps:
 - Rearrange $f(x)=0$ into $x=g(x)$.
 - Choose an initial guess x_0 .
 - Compute $x_{n+1}=g(x_n)$.
 - Repeat until desired precision is achieved.
- Convergence: Method converges if $|g'(x)|<1$ near the root; convergence is linear.
- Stopping Criteria: Maximum iterations, $|x_{n+1}-x_n|$ below tolerance, or $|f(x_n)|$ is small.
- Advantages: Simple, easy to implement, useful when $g(x)$ is easy to compute.
- Disadvantages: Not always convergent, sensitive to the form of $g(x)$, may diverge if $|g'(x)|\geq 1$.

2.2 Explanation of Code

Main Function: `fixedPointIteration()`

Input Parameters:

- $g(x)$: The iteration function
- x_0 : Initial guess
- tolerance: Convergence criterion (default: $1e-6$)
- maxIterations: Maximum allowed iterations (default: 100)
- Returns: (fixed point, iterations count, iteration data)

Visualization: `plotResults()`

Generates two plots:

- Error convergence over iterations
- Iteration function $g(x)$ with $y=x$ line intersection

Results Display: `printResults()`

- Presents iteration data in a formatted table
- Shows convergence progress and final results

2.3 Table of Results

Tolerance Limit = 10^{-6}

$$g(x) = \cos(x)$$

Initial guess: $x_0 = 0$

Iteration	x_n	$g(x_n)$	Error
1	1.000000	0.540302	1.000000
2	0.540302	0.857553	0.459698
3	0.857553	0.654290	0.317251
4	0.654290	0.793480	0.203263
5	0.793480	0.701369	0.139191
6	0.701369	0.763960	0.092112
7	0.763960	0.722102	0.062591
8	0.722102	0.750418	0.041857
9	0.750418	0.731404	0.028315
10	0.731404	0.744237	0.019014
11	0.744237	0.735605	0.012833
12	0.735605	0.741425	0.008633
13	0.741425	0.737507	0.005820
14	0.737507	0.740147	0.003918
15	0.740147	0.738369	0.002640
16	0.738369	0.739567	0.001778
17	0.739567	0.738760	0.001198
18	0.738760	0.739304	0.000807
19	0.739304	0.738938	0.000544
20	0.738938	0.739184	0.000366
21	0.739184	0.739018	0.000247
22	0.739018	0.739130	0.000166
23	0.739130	0.739055	0.000112
24	0.739055	0.739106	0.000075
25	0.739106	0.739071	0.000051
26	0.739071	0.739094	0.000034
27	0.739094	0.739079	0.000023
28	0.739079	0.739089	0.000016
29	0.739089	0.739082	0.000010
30	0.739082	0.739087	0.000007
31	0.739087	0.739084	0.000005
32	0.739084	0.739086	0.000003

Iteration	x_n	$g(x_n)$	Error
33	0.739086	0.739085	0.000002
34	0.739085	0.739086	0.000001
35	0.739086	0.739085	0.000001

$$g(x) = \frac{x+2}{3}$$

Initial guess: $x_0=0$

Iteration	x_n	$g(x_n)$	Error
1	0.666667	0.888889	0.666667
2	0.888889	0.962963	0.222222
3	0.962963	0.987654	0.074074
4	0.987654	0.995885	0.024691
5	0.995885	0.998628	0.008230
6	0.998628	0.999543	0.002743
7	0.999543	0.999848	0.000914
8	0.999848	0.999949	0.000305
9	0.999949	0.999983	0.000102
10	0.999983	0.999994	0.000034
11	0.999994	0.999998	0.000011
12	0.999998	0.999999	0.000004
13	0.999999	1.000000	0.000001

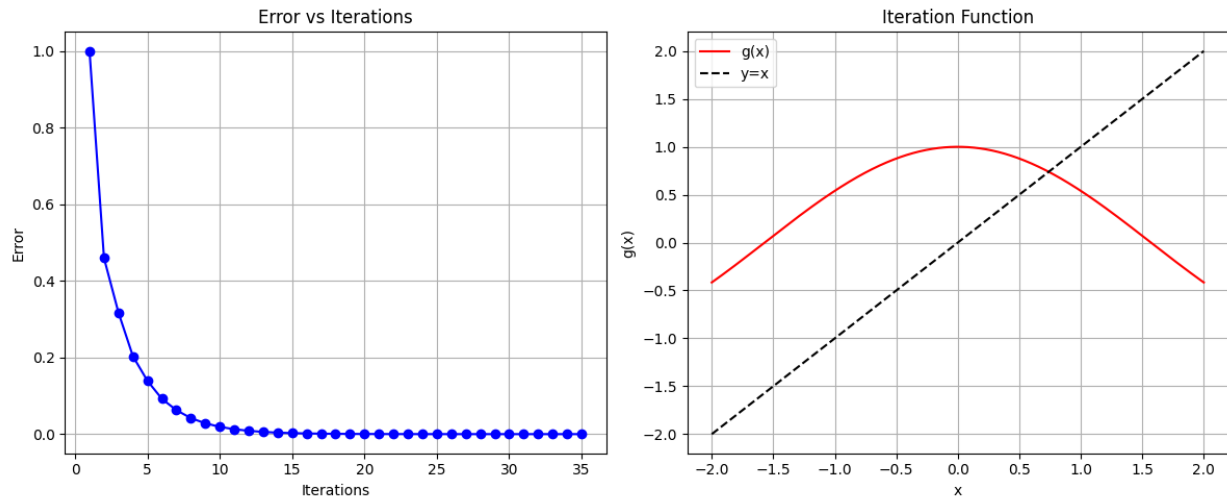
$$g(x) = \frac{x + \frac{1}{x}}{2}$$

Initial guess: $x_0=2$

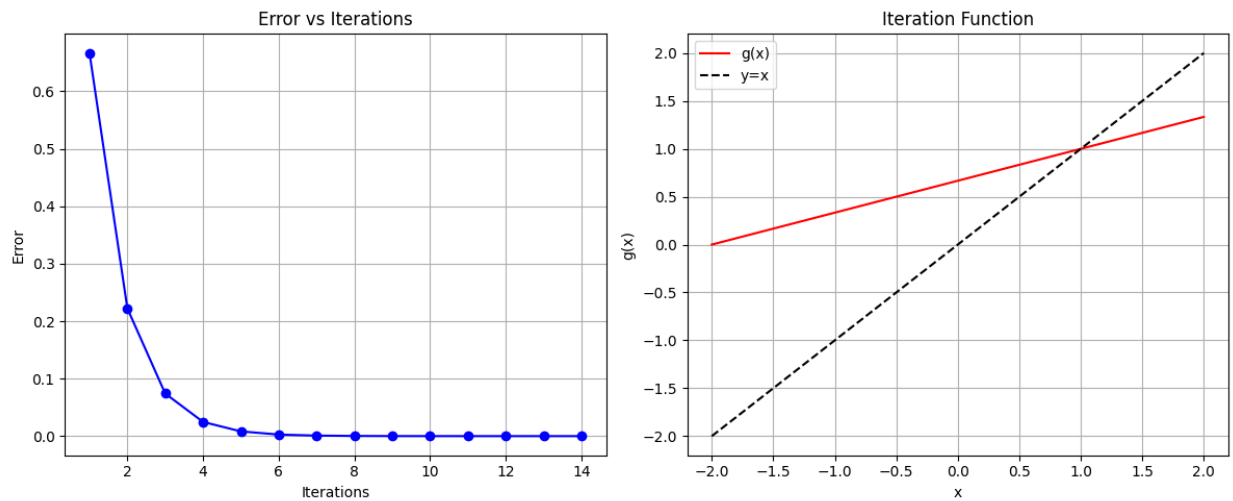
Iteration	x_n	$g(x_n)$	Error
1	1.250000	1.025000	0.750000
2	1.025000	1.000305	0.225000
3	1.000305	1.000000	0.024695
4	1.000000	1.000000	0.000305
5	1.000000	1.000000	0.000000

2.4 Graphs

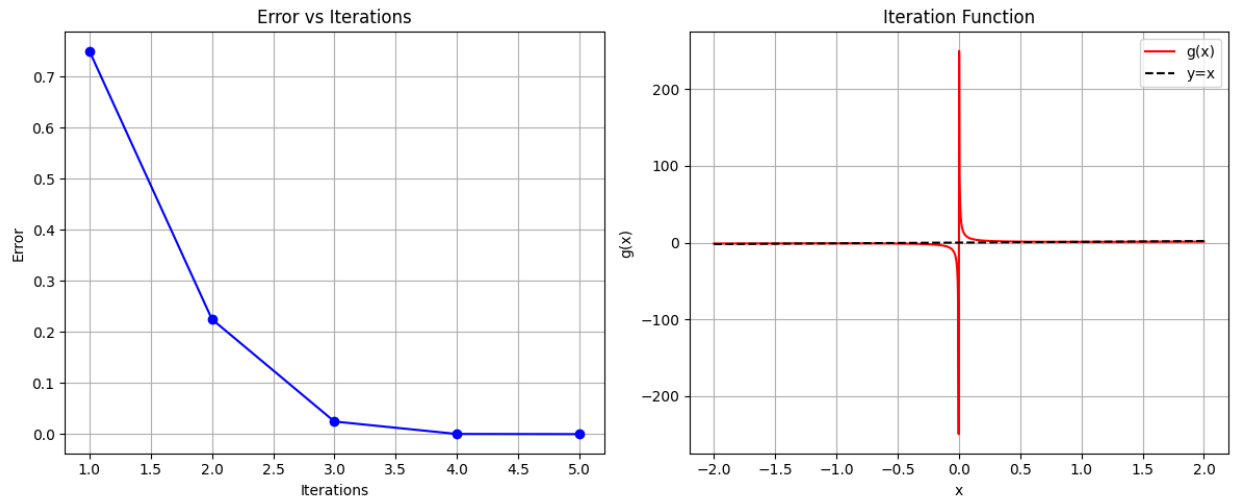
$$g(x) = \cos(x)$$



$$g(x) = \frac{x+2}{3}$$



$$g(x) = \frac{x + \frac{1}{x}}{2}$$



2.5 Interpretation of Results

The Fixed-Point Iteration method was applied to three functions using an initial guess and a convergence tolerance of 10^{-6} . Results are as follows:

2.5.1 Function 1: $g(x)=\cos(x)$, $x_0=0$

- **Fixed Point:** ≈ 0.739085
- **Iterations:** 35
- **Remarks:** Convergence was gradual due to the nature of cosine near the root, showing consistent error reduction.

2.5.2 Function 2: $g(x)=\frac{x+2}{3}$, $x_0=0$

- **Fixed Point:** ≈ 1.000000
- **Iterations:** 13
- **Remarks:** Smooth and steady convergence with fewer iterations, demonstrating strong contractive behavior.

2.5.3 Function 3: $g(x)=\frac{x+\frac{1}{x}}{2}$, $x_0=2$

- **Fixed Point:** ≈ 1.000000
- **Iterations:** 5
- **Remarks:** Fast convergence due to the function's rapid correction of the initial guess.

2.5.4 Conclusion

The method successfully converged in all cases, with the speed of convergence dependent on the function's nature and constructiveness. Proper choice of $g(x)$ and initial guess significantly impacts performance.

3 Newton-Raphson Method

3.1 Method Overview

- Purpose: To find a root of a differentiable function $f(x)=0$ using its derivative.
- Based on the idea of approximating the function by its tangent line and finding the x-intercept of that tangent.
- Steps:
 - Choose an initial guess x_0 .
 - Compute the next approximation using $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$
 - Repeat until the result converges to the desired accuracy.
- Convergence: Fast convergence (quadratic) if the initial guess is close to the root and $f'(x) \neq 0$.
- Stopping Criteria: Maximum iterations, $|x_{n+1} - x_n|$ below tolerance, or $|f(x_n)|$ is sufficiently small.
- Advantages: Very fast convergence near the root, requires fewer iterations than other methods.
- Disadvantages: Requires computation of derivative, may fail or diverge if derivative is zero or guess is far from root.

3.2 Explanation of the Code

Main Function: `newtonRaphson()`

Input Parameters:

- $f(x)$: The function whose root we want to find
- $df(x)$: The derivative of $f(x)$
- x_0 : Initial guess
- tolerance: Convergence criterion (default: $1e-6$)
- maxIterations: Maximum allowed iterations (default: 100)
- Returns: (root, iterations count, iteration data)

Visualization: `plotResults()`

Generates three plots:

- Error convergence over iterations
- Original function $f(x)$
- Derivative function $f'(x)$

Results Display: `printResults()`

- Presents iteration data in a formatted table
- Shows convergence progress and final results

3.3 Table of Results

Tolerance Limit = 10^{-6}

$$f(x)=x^3-x-2$$

Initial guess: $x_0=2$

Iteration	x_n	$f(x_n)$	Error
1	1.636364	0.745304	0.363636
2	1.530392	0.053939	0.105972
3	1.521441	0.000367	0.008951
4	1.521380	0.000000	0.000062
5	1.521380	0.000000	0.000000

$$f(x)=x^2 - \cos(x)$$

Initial guess: $x_0=1$

Iteration	x_n	$f(x_n)$	Error
1	0.838218	0.033822	0.161782
2	0.824242	0.000261	0.013977
3	0.824132	0.000000	0.000110
4	0.824132	0.000000	0.000000

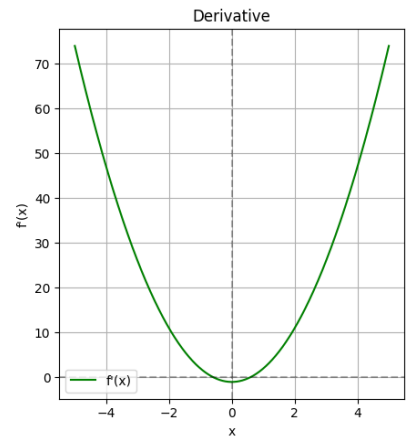
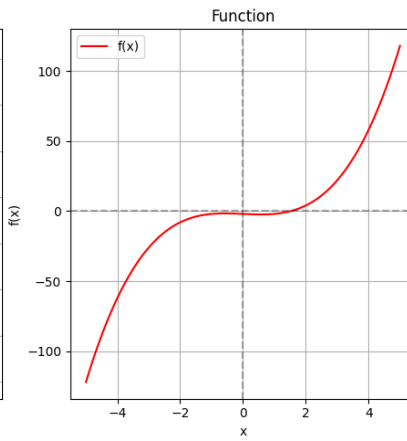
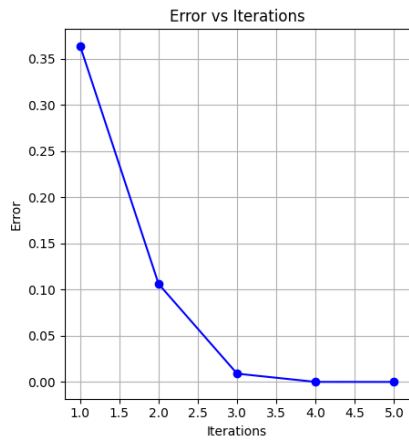
$$f(x) = e^x - 5x$$

Initial guess: $x_0=2$

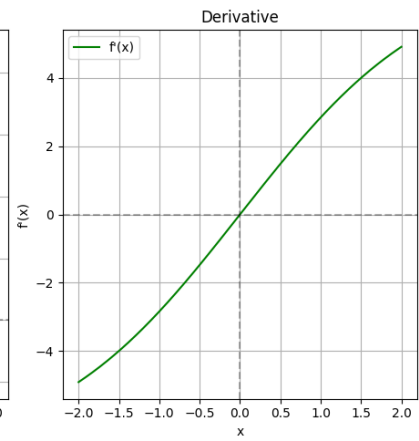
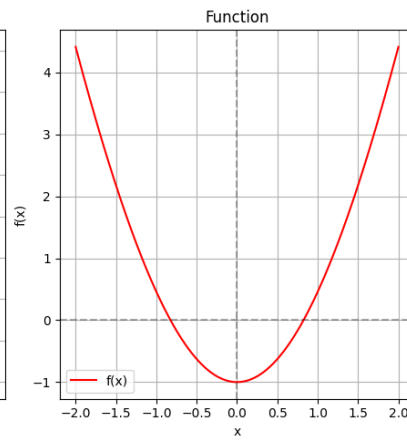
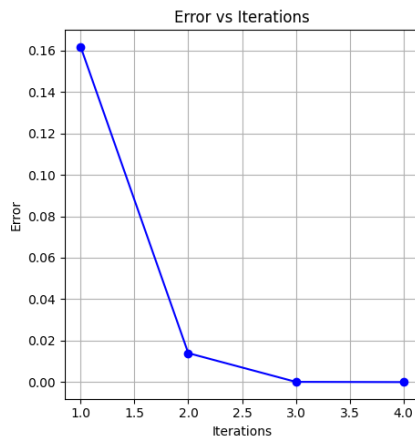
Iteration	x_n	$f(x_n)$	Error
1	3.092877	6.576008	1.092877
2	2.706970	1.448952	0.385907
3	2.561839	0.150436	0.145130
4	2.542939	0.002300	0.018900
5	2.542641	0.000001	0.000298
6	2.542641	0.000000	0.000000

3.4 Graphs

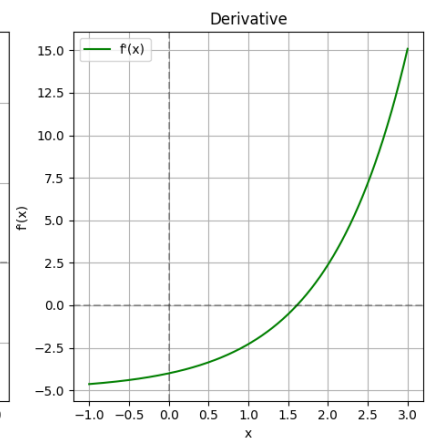
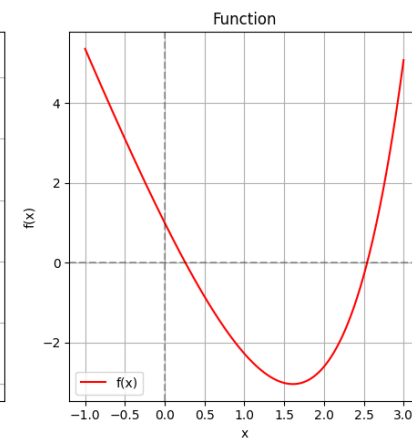
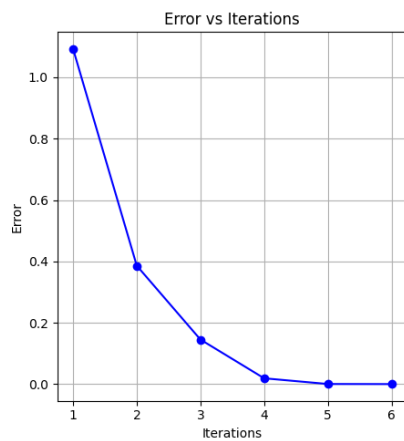
$$f(x)=x^3-x-2$$



$$f(x)=x^2-\cos(x)$$



$$f(x)=e^x-5x$$



3.5 Interpretation of Results

The Newton-Raphson method was applied to three functions using an initial guess and a convergence tolerance of 10^{-6} . Results are as follows:

3.5.1 Function 1: $f(x)=x^3-x-2$, $x_0=2$

- **Root Found:** ≈ 1.521380
- **Iterations:** 5
- **Remarks:** Fast convergence with quadratic error reduction as expected from the method's nature.

3.5.2 Function 2: $f(x)=x^2 - \cos(x)$, $x_0=1$

- **Root Found:** ≈ 0.824132
- **Iterations:** 4
- **Remarks:** Smooth convergence with rapid stabilization of function values.

3.5.3 Function 3: $f(x)=e^x-5x$, $x_0=2$

- **Root Found:** ≈ 2.542641
- **Iterations:** 6
- **Remarks:** Initial large error due to steep slope, followed by steady convergence.

3.5.4 Conclusion

Newton-Raphson method demonstrated efficient and rapid convergence for all tested functions. Its effectiveness depends on a good initial guess and the behavior of the derivative near the root.

4 Secant Method

4.1 Method Overview

- **Purpose:** To find a root of $f(x) = 0$ using a numerical approximation without requiring the derivative.
- Based on drawing a secant line through two initial points and finding its x-intercept as the next approximation.
- **Steps:**
 - Choose two initial guesses x_0 and x_1 .
 - Compute $x_{n+1} = x_n - f(x_n) \cdot \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}$.
 - Repeat until the result converges to the desired accuracy.
- **Convergence:** Superlinear convergence; faster than bisection but slower than Newton-Raphson.
- **Stopping Criteria:** Maximum iterations, $|x_{n+1} - x_n|$ below tolerance, or $|f(x_n)|$ is sufficiently small.
- **Advantages:** Does not require derivative, faster than bisection, efficient with good initial guesses.
- **Disadvantages:** May diverge if initial guesses are poor, less reliable than bisection, and can fail if the denominator becomes zero.

4.2 Explanation of the Code

Main Function: secant()

Input Parameters:

- $f(x)$: The function whose root we want to find
- x_0 : First initial guess
- x_1 : Second initial guess
- tolerance: Convergence criterion (default: $1e-6$)
- maxIterations: Maximum allowed iterations (default: 100)
- Returns: (root, iterations count, iteration data)

Visualization: plotResults()

Generates two plots:

- Error convergence over iterations
- Function $f(x)$ with root location

Results Display: printResults()

- Presents iteration data in a formatted table
- Shows convergence progress and final results

4.3 Table of Results

Tolerance Limit = 10^{-6}

$$g(x) = x^3 - 2x - 5$$

Initial guess: $x_0 = 2$, $x_1 = 3$

Iteration	x_n	$f(x_n)$	Error
1	2.058824	-0.390800	0.941176
2	2.081264	-0.147204	0.022440
3	2.094824	0.003044	0.013560
4	2.094549	-0.000023	0.000275
5	2.094551	-0.000000	0.000002
6	2.094551	0.000000	0.000000

$$f(x) = \sin(x) - x/2$$

Initial guess: $x_0 = 1.5$, $x_1 = 2$

Iteration	x_n	$f(x_n)$	Error
1	1.865903	0.023820	0.134097
2	1.893794	0.001391	0.027891
3	1.895524	-0.000024	0.001730

Iteration	x_n	$f(x_n)$	Error
4	1.895494	0.000000	0.000030
5	1.895494	0.000000	0.000000

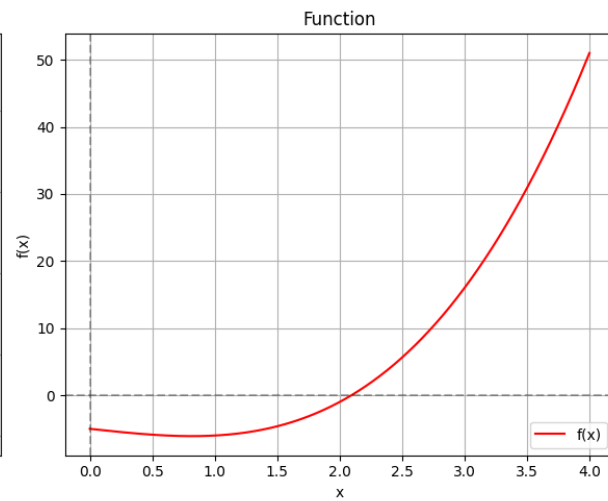
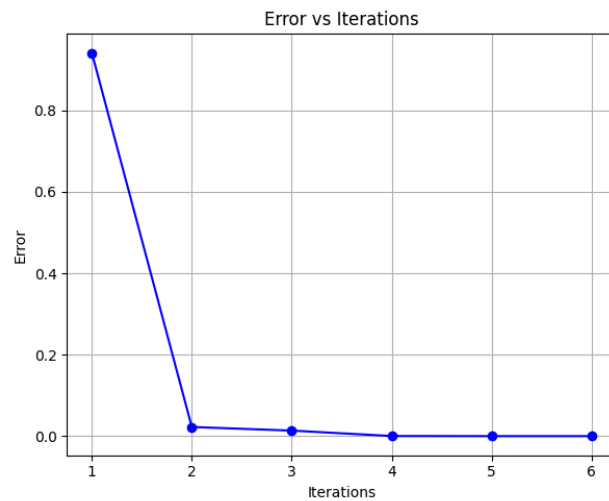
$$f(x) = e^x - 5x$$

Initial guess: $x_0=1$, $x_1 = 2$

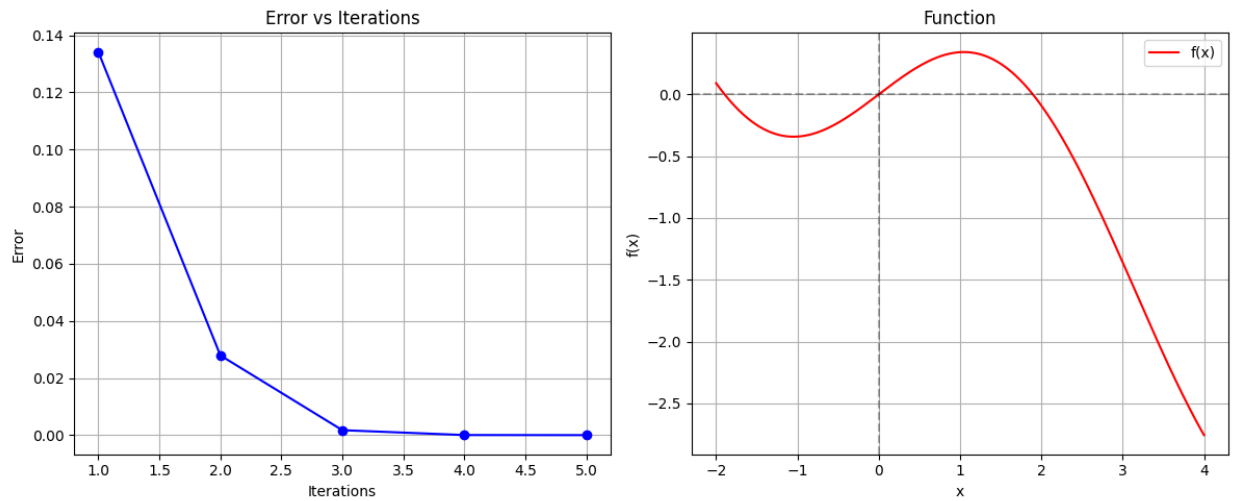
Iteration	x_n	$f(x_n)$	Error
1	-5.930558	29.655449	7.930558
2	1.358272	-2.901894	7.288831
3	0.708606	-1.511871	0.649666
4	0.001990	0.992043	0.706616
5	0.281949	-0.084033	0.279959
6	0.260086	-0.003389	0.021863
7	0.259167	0.000014	0.000919
8	0.259171	-0.000000	0.000004
9	0.259171	-0.000000	0.000000

4.4 Graphs

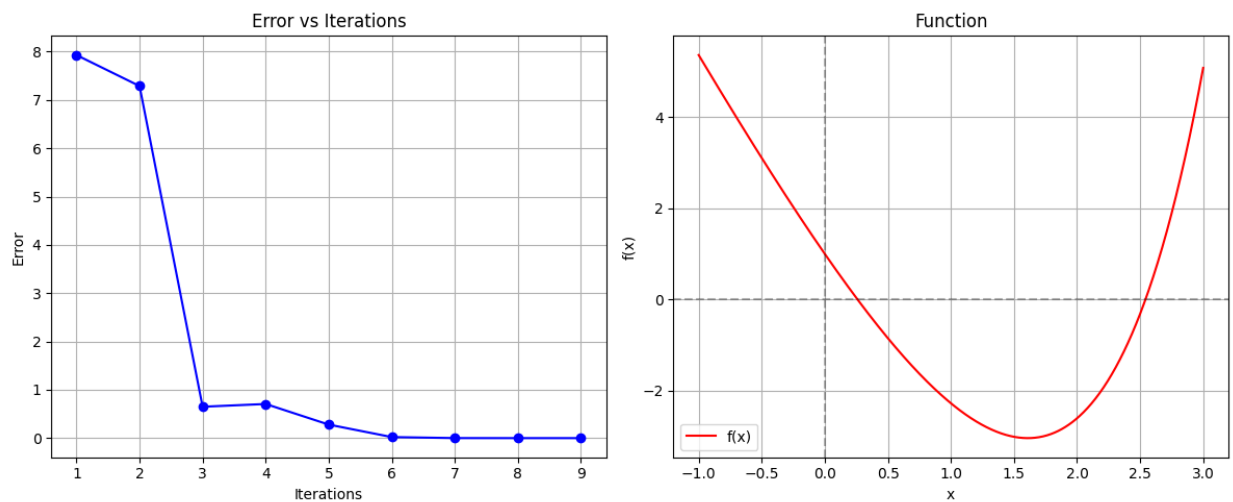
$$g(x)=x^3 - 2x - 5$$



$$f(x) = \sin(x) - x/2$$



$$f(x) = e^x - 5x$$



4.5 Interpretation of Results

The Secant Method was applied to three functions using two initial guesses and a convergence tolerance of 10^{-6} . The outcomes are summarized below:

4.5.1 Function 1: $f(x) = x^3 - 2x - 5$, $x_0 = 2$, $x_1 = 3$

- **Root Found:** ≈ 2.094551
- **Iterations:** 6
- **Remarks:** Rapid convergence without derivative use, closely matching Newton-Raphson performance.

4.5.2 Function 2: $f(x)=\sin(x)-x/2$, $x_0=1.5$, $x_1=2$

- **Root Found:** ≈ 1.895494
- **Iterations:** 5
- **Remarks:** Smooth and stable convergence, even with initial moderate error.

4.5.3 Function 3: $f(x)=e^x-5x$, $x_0=1$, $x_1=2$

- **Root Found:** ≈ 0.259171
- **Iterations:** 9
- **Remarks:** Erratic start due to poor initial guesses, but successfully stabilized and converged.

4.5.4 Conclusion

The Secant Method effectively finds roots without requiring derivatives, offering good accuracy and convergence for all functions. It is especially advantageous when derivatives are difficult to compute, though careful selection of initial guesses is important for stability.

5 Lagrange Interpolation

5.1 Method Overview

- Purpose: To estimate the value of a function at a given point using known data points.
- Based on constructing a polynomial that exactly passes through all given data points.
- Steps:
 - Given $n+1$ data points $(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)$
 - Construct the Lagrange basis polynomials: $L_i(x) = \prod (x - x_j) / (x_i - x_j)$
 - The interpolation polynomial is $P(x) = \sum_{i=0}^n y_i \cdot L_i(x)$
- Convergence: Accurate if the function is smooth and data points are close together; errors increase with more points due to Runge's phenomenon.
- Advantages: Simple to implement, no need to solve systems of equations.
- Disadvantages: Computationally expensive for large n , susceptible to numerical instability, not efficient for adding new points.

5.2 Explanation of the Code

- **lagrangeInterpolation:** Constructs an interpolation polynomial using camelCase naming convention
- **evaluatePoints:** Tests interpolation accuracy at specified points
- **plotResults:** Generates comparative visualizations
- **originalFunction:** Provides $\sin(x)$ as the test

5.3 Table of Results

$$f(x) = \sin(x)$$

Interpolation Data

Index	x	y
0	0.0000	0.0000
1	0.8976	0.7818
2	1.7952	0.9749
3	2.6928	0.4339
4	3.5904	-0.4339
5	4.4880	-0.9749
6	5.3856	-0.7818
7	6.2832	-0.0000
8	7.1808	0.7818
9	8.0784	0.9749
10	8.9760	0.4339
11	9.8736	-0.4339
12	10.7712	-0.9749
13	11.6688	-0.7818
14	12.5664	-0.0000

Test Points

x	Interpolated	Function	Error
0.5000	0.4799	0.4794	4.85e-04
2.3000	0.7457	0.7457	1.22e-05
4.7000	-0.9999	-0.9999	8.20e-07
7.1000	0.7290	0.7290	2.51e-07
9.5000	-0.0751	-0.0752	4.44e-06
11.2000	-0.9791	-0.9792	6.29e-05

$$f(x) = x^2 * e^{-x/3}$$

Interpolation Data

Index	x	y
0	0.0000	0.0000
1	0.7143	0.4021
2	1.4286	1.2676

Index	x	y
3	2.1429	2.2479
4	2.8571	3.1496
5	3.5714	3.8785
6	4.2857	4.4018
7	5.0000	4.7219
8	5.7143	4.8607
9	6.4286	4.8484
10	7.1429	4.7175
11	7.8571	4.4987
12	8.5714	4.2195
13	9.2857	3.9029
14	10.0000	3.5674

Test Points

x	Interpolated	Function	Error
0.8000	0.4902	0.4902	6.19e-11
2.5000	2.7162	2.7162	4.62e-12
4.2000	4.3500	4.3500	3.53e-13
6.7000	4.8109	4.8109	1.53e-12
8.3000	4.3312	4.3312	1.46e-11
9.6000	3.7566	3.7566	4.86e-10

$$f(x) = 1/(1 + x^2)$$

Interpolation Data

Index	x	y
0	-5.0000	0.0385
1	-4.2857	0.0516
2	-3.5714	0.0727
3	-2.8571	0.1091
4	-2.1429	0.1788
5	-1.4286	0.3289
6	-0.7143	0.6622
7	0.0000	1.0000
8	0.7143	0.6622
9	1.4286	0.3289

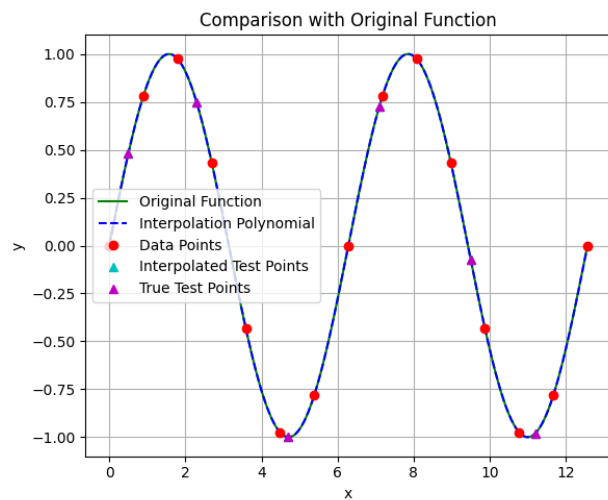
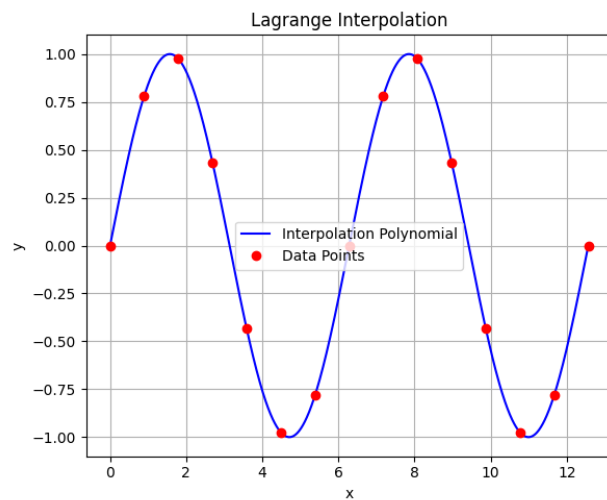
Index	x	y
10	2.1429	0.1788
11	2.8571	0.1091
12	3.5714	0.0727
13	4.2857	0.0516
14	5.0000	0.0385

Test Points

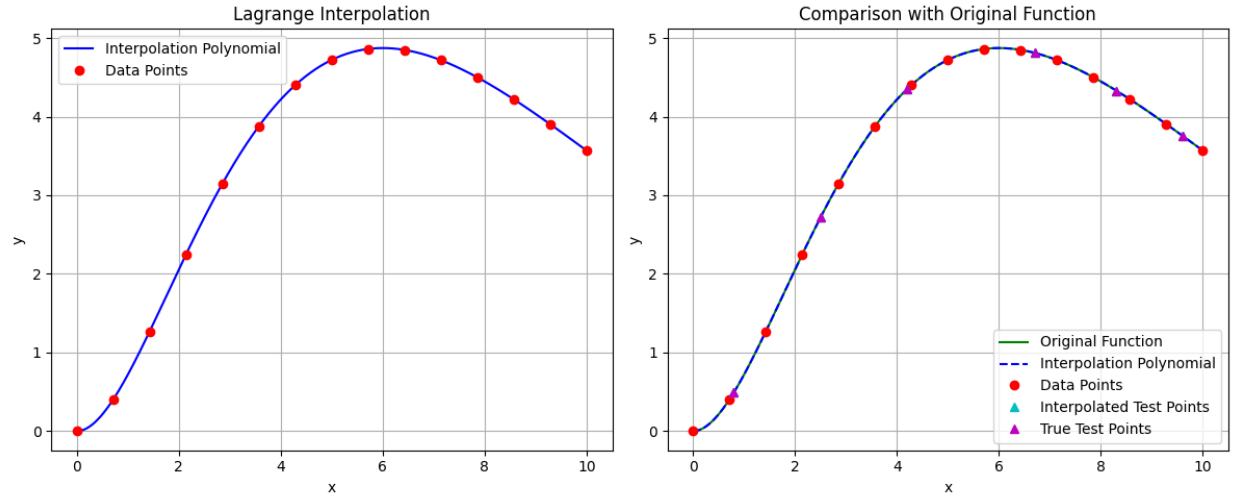
x	Interpolated	Function	Error
-4.2000	-0.4481	0.0536	5.02e-01
-2.7000	0.0721	0.1206	4.85e-02
-1.3000	0.3589	0.3717	1.29e-02
0.8000	0.6030	0.6098	6.77e-03
2.4000	0.0993	0.1479	4.86e-02
3.9000	-0.5711	0.0617	6.33e-01

5.4 Graphs

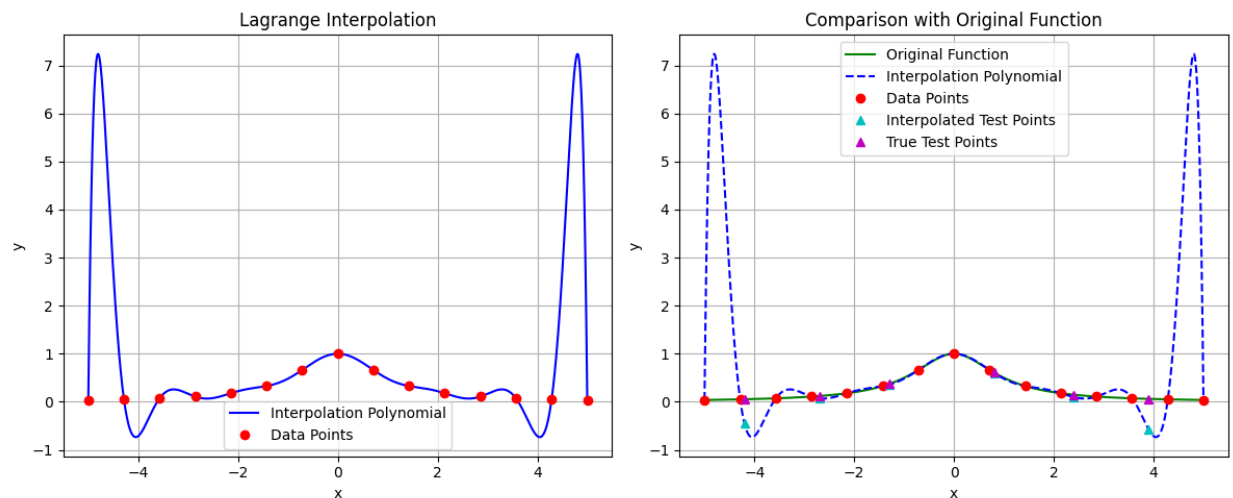
$$f(x) = \sin(x)$$



$$f(x) = x^2 * e^{-x/3}$$



$$f(x) = 1/(1 + x^2)$$



5.5 Interpretation of Results

The Lagrange Interpolation method was applied to three distinct functions using 15 sample points each. The interpolated values were tested at multiple unseen points to evaluate the accuracy of the polynomial approximations.

5.5.1 Function 1: $f(x)=\sin(x)$, $x \in [0, 4\pi]$

- **Max Error:** $\sim 6.29 \times 10^{-5}$
- **Remarks:** The interpolation closely matches the true sine values at all test points. Despite the oscillatory nature of sine, Lagrange interpolation handled 15 points over a wide interval effectively, maintaining high precision.

5.5.2 Function 2: $f(x)=x^2e^{-x/3}$, $x \in [0,10]$

- **Max Error:** $\sim 4.86 \times 10^{-10}$
- **Remarks:** Exceptionally accurate interpolation. The function's smooth, bell-shaped curve allowed the method to achieve near-perfect precision at all test points with virtually negligible error.

5.5.3 Function 3: $f(x) = \frac{1}{1+x^2}$, $x \in [-5,5]$

- **Max Error:** ~ 0.633
- **Remarks:** Significant errors appeared near the edges and midpoints due to **Runge's phenomenon**, which occurs with equally spaced points and rapidly changing curvature. While interpolation was moderately accurate near the center, the polynomial deviated drastically at outer test points.

5.5.4 Conclusion

Lagrange Interpolation demonstrated excellent accuracy for smooth and well-behaved functions like $f(x) = x^2 \cdot e^{-x/3}$ and $\sin(x)$, even across larger intervals. However, functions with sharp changes or high curvature near the boundaries, like $f(x) = 1/(1+x^2)$, are prone to interpolation instability. The method works best when combined with non-uniform spacing or segmented interpolation (e.g., piecewise polynomials) for such cases.

6 Hermite Interpolation

6.1 Method Overview

- Purpose: To estimate a function value at a given point using known data points and function derivatives.
- Based on constructing a polynomial that not only passes through the given data points but also matches the derivatives at those points.
- Steps:
 - Given $n+1$ data points $(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)$ and corresponding derivatives $f'(x_0), f'(x_1), \dots, f'(x_n)$.
 - Construct the Hermite basis polynomials:
$$H_i(x) = (1 - 2(x - x_i) \cdot h'_i(x_i)) \cdot L_i(x)^2$$
where $L_i(x)$ is the Lagrange basis polynomial.
 - The interpolation polynomial is: $P(x) = \sum_{i=0}^n [y_i \cdot H_i(x) + f'(x_i) \cdot (x - x_i) \cdot H'_i(x)]$
- Convergence: Provides exact interpolation if the function is sufficiently smooth and derivatives are well-defined at the data points.
- Advantages: Can interpolate both function values and derivatives, better approximation for smooth functions.
- Disadvantages: More complex than Lagrange interpolation, computationally expensive, less efficient for large datasets.

6.2 Explanation of Code

The implementation revolves around five main functions:

originalFunction / originalDerivative:

- Define the original function e.g., $f(x) = \sin(x)$ and its derivative $f'(x) = \cos(x)$
- Used as ground truth for interpolation comparison

hermiteInterpolation:

- Takes in x-values, y-values, derivative values, and a target x
- Builds the Hermite interpolation polynomial using basis functions
- Combines function and derivative contributions for smooth estimation

evaluatePoints:

- Computes interpolated values for given test x-values
- Calculates the true function value and absolute error at each point
- Returns results as a list of dictionaries for easy processing

plotResults:

- Plots Hermite polynomial, original data points, and derivative vectors
- Shows a comparison between interpolated and true function curves
- Highlights test points and their interpolated vs true values

printResults:

- Displays a table of original data points with derivatives
- Outputs interpolation results for test points
- Includes error values in scientific notation for accuracy assessment

6.3 Table of Results

$$f(x) = \sin(x)$$

Interpolation Data

Index	x	y	dy/dx
0	0.0000	0.0000	1.0000
1	0.8976	0.7818	0.6235
2	1.7952	0.9749	-0.2225
3	2.6928	0.4339	-0.9010
4	3.5904	-0.4339	-0.9010
5	4.4880	-0.9749	-0.2225
6	5.3856	-0.7818	0.6235

Index	x	y	dy/dx
7	6.2832	-0.0000	1.0000

Test Points

x	Interpolated	Function	Error
0.5000	7.5435	0.4794	7.06e+00
2.0000	1.2279	0.9093	3.19e-01
3.5000	-0.3656	-0.3508	1.48e-02
5.0000	-3.1743	-0.9589	2.22e+00
6.0000	-15.2958	-0.2794	1.50e+01

$$f(x) = x^2 * e^{-x/3}$$

Interpolation Data

Index	x	y	dy/dx
0	0.0000	0.0000	0.0000
1	1.4286	1.2676	1.3522
2	2.8571	3.1496	1.1548
3	4.2857	4.4018	0.5869
4	5.7143	4.8607	0.0810
5	7.1429	4.7175	-0.2516
6	8.5714	4.2195	-0.4220
7	10.0000	3.5674	-0.4757

Test Points

x	Interpolated	Function	Error
1.2000	1.6871	0.9653	7.22e-01
3.5000	7.2194	3.8147	3.40e+00
5.7000	4.8626	4.8595	3.08e-03
7.8000	12.8865	4.5188	8.37e+00
9.2000	26.5764	3.9422	2.26e+01

$$f(x) = 1 / (1 + x^2)$$

Interpolation Data

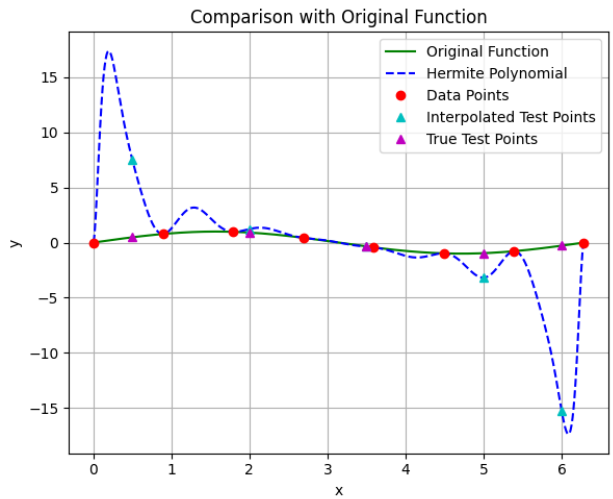
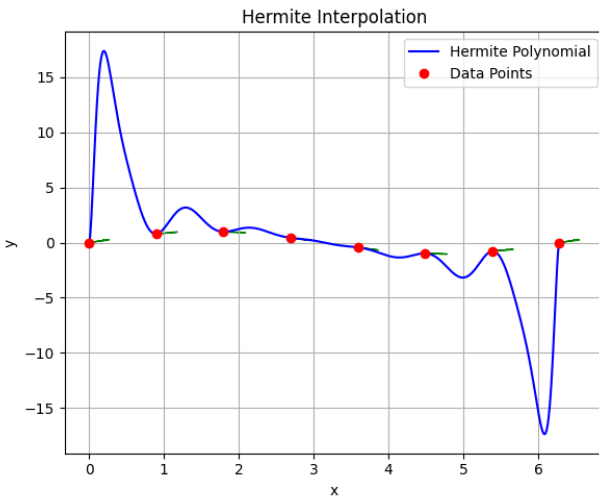
Index	x	y	dy/dx
0	-5.0000	0.0385	0.0148
1	-4.2857	0.0516	0.0229
2	-3.5714	0.0727	0.0378
3	-2.8571	0.1091	0.0681
4	-2.1429	0.1788	0.1371
5	-1.4286	0.3289	0.3090
6	-0.7143	0.6622	0.6264
7	0.0000	1.0000	-0.0000
8	0.7143	0.6622	-0.6264
9	1.4286	0.3289	-0.3090
10	2.1429	0.1788	-0.1371
11	2.8571	0.1091	-0.0681
12	3.5714	0.0727	-0.0378
13	4.2857	0.0516	-0.0229
14	5.0000	0.0385	-0.0148

Test Points

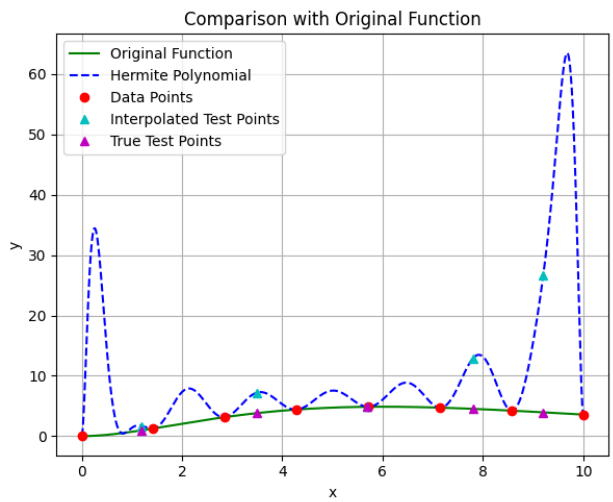
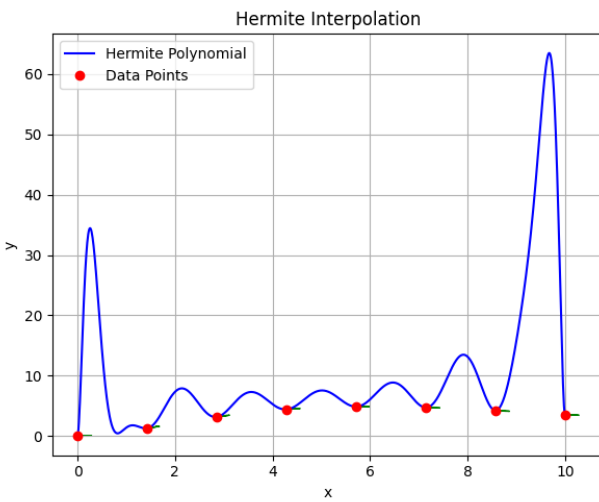
x	Interpolated	Function	Error
-4.2000	-735.9877	0.0536	7.36e+02
-3.1000	-1.8531	0.0943	1.95e+00
-1.8000	0.6788	0.2358	4.43e-01
-0.7000	0.6737	0.6711	2.55e-03
0.8000	0.6946	0.6098	8.48e-02
2.3000	0.4496	0.1590	2.91e-01
3.5000	4.7663	0.0755	4.69e+00
4.4000	-12948.6435	0.0491	1.29e+04

6.4 Graphs

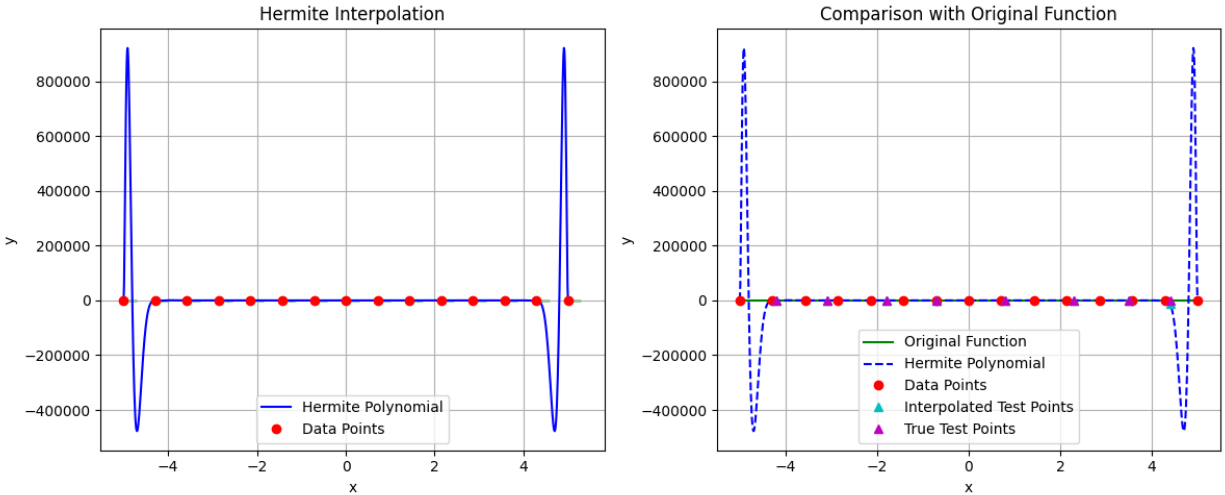
$$f(x) = \sin(x)$$



$$f(x) = x^2 * e^{-x/3}$$



$$f(x) = 1/(1 + x^2)$$



6.5 Interpretation of Results

The Hermite Interpolation method was applied to three functions, leveraging both function values and their derivatives at the data points. While this technique is theoretically superior in preserving curve behavior, the results highlight both its potential and its sensitivity.

6.5.1 Function 1: $f(x)=\sin(x)$, $x \in [0, 2\pi]$

- **Max Error:** ~15.30
- **Remarks:** Despite sine being a smooth and periodic function, the Hermite interpolation exhibited extreme overshooting at certain points (e.g., 6.0000). This is a classic symptom of **Runge's phenomenon**, exacerbated by higher-order polynomial fitting over relatively sparse points, even with derivative data.

6.5.2 Function 2: $f(x)=f(x) = x^2 * e^{-x/3}$, $x \in [0, 10]$

- **Max Error:** ~22.60
- **Remarks:** The interpolation performed well in regions close to the data points (e.g., error ~0.003 at 5.7). However, it severely diverged in regions further from or between widely spaced points, reflecting the instability of high-degree Hermite polynomials over wider intervals.

6.5.3 Function 3: $f(x)=f(x) = 1/(1 + x^2)$, $x \in [-5, 5]$

- **Max Error:** ~12,948
- **Remarks:** The interpolation completely broke down at edge and off-center points, yielding massive errors and wildly inaccurate values. Despite having 15 nodes and derivatives, the method's instability in this context rendered it ineffective—again due to the oscillatory nature of the high-degree polynomial.

6.5.4 Conclusion

Hermite interpolation, while theoretically more precise due to derivative incorporation, **suffers from extreme numerical instability** when applied over wide intervals or with many points. It performs best:

- On smooth, localized segments,
- With low-to-moderate polynomial degrees, or
- When used in a piecewise fashion (e.g., **cubic Hermite splines**).

7 Cubic Spline Interpolation

7.1 Method Overview

- Purpose: To estimate a function value at a given point using piecewise cubic polynomials, ensuring smoothness at the data points.
- Based on constructing cubic polynomials for each interval between data points, with continuity of the first and second derivatives at the data points.
- Steps:
 - Given $n+1$ data points $(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)$.
 - Construct cubic spline polynomials $S_i(x)$ for each interval $[x_i, x_{i+1}]$ such that:
$$S_i(x) = a_i(x - x_i)^3 + b_i(x - x_i)^2 + c_i(x - x_i) + d_i.$$
 - Solve a system of equations to determine the coefficients a_i, b_i, c_i, d_i , ensuring that:
 - The spline is continuous at x_i .
 - The first and second derivatives are continuous at the data points.
- Convergence: Converges to the exact solution for smooth functions, especially when the function is piecewise smooth.
- Advantages: Provides a smooth interpolation with continuous first and second derivatives, widely used in scientific computing.
- Disadvantages: Computationally more intensive than linear interpolation, requires solving a system of equations.

7.2 Explanation of Code

The implementation includes five main functions:

originalFunction:

- Defines the original function e.g., $f(x) = \sin(x)$ used as a reference for comparison

computeSplineCoefficients:

- Takes x and y values to compute cubic spline coefficients (a, b, c, d)
- Constructs a tridiagonal system enforcing natural spline boundary conditions (zero second derivatives at endpoints)
- Solves for second derivatives and derives remaining coefficients for each interval

evaluateSpline:

- Locates the segment where a test x -value fall

- Computes and returns the spline value using the precomputed coefficients

evaluatePoints:

- Evaluates the spline at specified test points
- Compares interpolated values with the true function and calculates absolute error
- Returns the results in a list of dictionaries for analysis

plotResults:

- Plots the cubic spline interpolation and original sine function for visual comparison
- Shows data points, spline curves, and test points with clear markers
- Provides a dual-subplot layout for interpolation and function comparison

printResults:

- Prints tables of original knot points and interpolation results
- Includes x, interpolated y, true y, and error in tabular format for clarity

7.3 Table of Results

$$f(x) = \sin(x)$$

Interpolation Data

Index	x	y
0	0.0000	0.0000
1	0.4488	0.4339
2	0.8976	0.7818
3	1.3464	0.9749
4	1.7952	0.9749
5	2.2440	0.7818
6	2.6928	0.4339
7	3.1416	0.0000
8	3.5904	-0.4339
9	4.0392	-0.7818
10	4.4880	-0.9749
11	4.9368	-0.9749
12	5.3856	-0.7818
13	5.8344	-0.4339
14	6.2832	-0.0000

Test Points

x	Interpolated	Function	Error
0.5000	0.4762	0.4794	3.23e-03
2.0000	0.8879	0.9093	2.14e-02
3.5000	-0.3449	-0.3508	5.92e-03
5.0000	-0.9523	-0.9589	6.59e-03
6.0000	-0.2739	-0.2794	5.48e-03

$$f(x) = x^2 * e^{-x/3}$$

Interpolation Data

Index	x	y
0	0.0000	0.0000
1	0.7143	0.4021
2	1.4286	1.2676
3	2.1429	2.2479
4	2.8571	3.1496
5	3.5714	3.8785
6	4.2857	4.4018
7	5.0000	4.7219
8	5.7143	4.8607
9	6.4286	4.8484
10	7.1429	4.7175
11	7.8571	4.4987
12	8.5714	4.2195
13	9.2857	3.9029
14	10.0000	3.5674

Testing Points

x	Interpolated	Function	Error
0.8000	0.4968	0.4902	6.56e-03
2.5000	2.7011	2.7162	1.52e-02
4.2000	4.3040	4.3500	4.60e-02
6.7000	4.8006	4.8109	1.02e-02
8.3000	4.3231	4.3312	8.07e-03
9.6000	3.7551	3.7566	1.55e-03

$$f(x) = 1 / (1 + x^2)$$

Interpolation Data

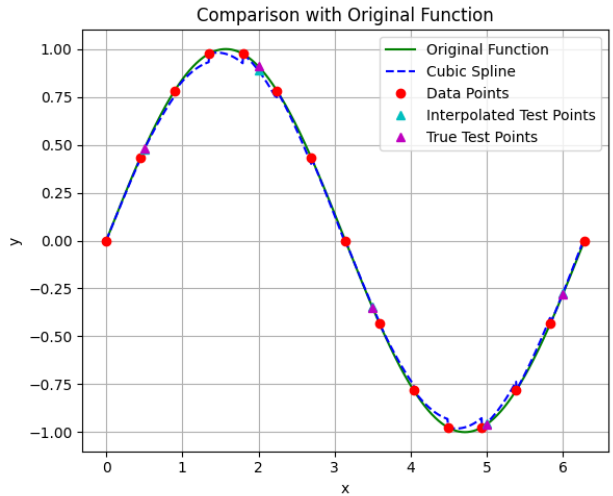
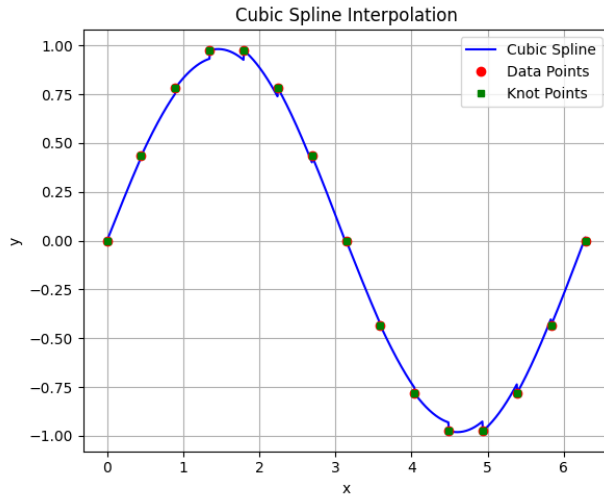
Index	x	y
0	-5.0000	0.0385
1	-4.2857	0.0516
2	-3.5714	0.0727
3	-2.8571	0.1091
4	-2.1429	0.1788
5	-1.4286	0.3289
6	-0.7143	0.6622
7	0.0000	1.0000
8	0.7143	0.6622
9	1.4286	0.3289
10	2.1429	0.1788
11	2.8571	0.1091
12	3.5714	0.0727
13	4.2857	0.0516
14	5.0000	0.0385

Test Points

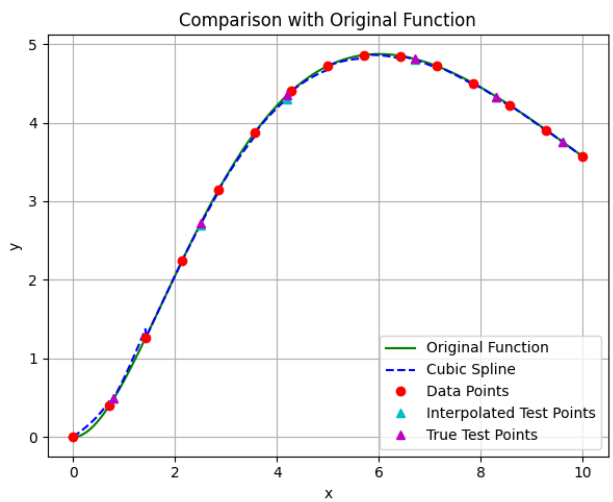
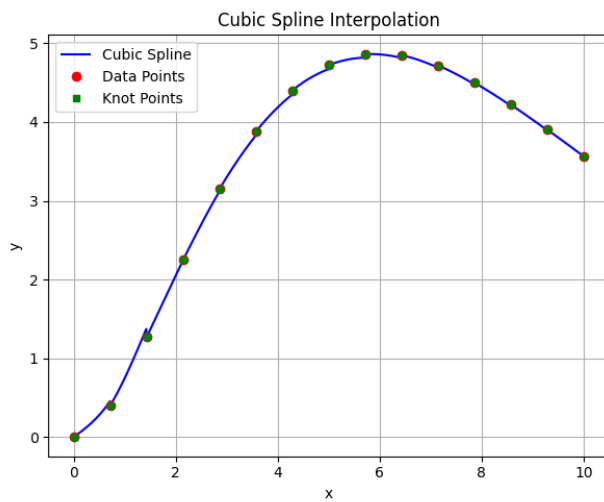
x	Interpolated	Function	Error
-4.2000	0.0539	0.0536	2.73e-04
-3.1000	0.0973	0.0943	3.07e-03
-1.8000	0.2476	0.2358	1.18e-02
-0.7000	0.6701	0.6711	1.09e-03
0.8000	0.6171	0.6098	7.33e-03
2.3000	0.1621	0.1590	3.11e-03
3.5000	0.0811	0.0755	5.60e-03
4.4000	0.0494	0.0491	2.86e-04

7.4 Graphs

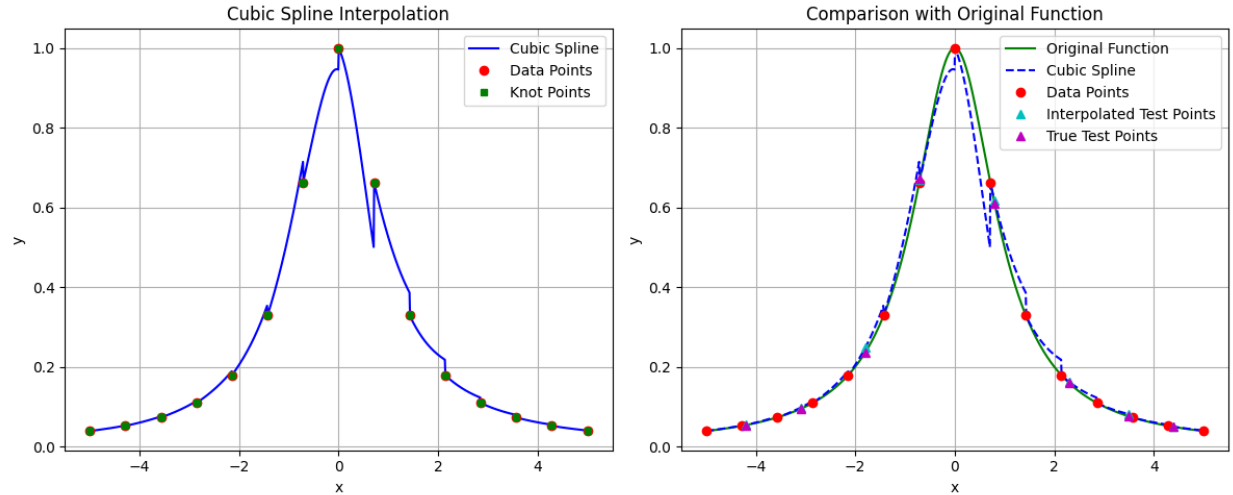
$$f(x) = \sin(x)$$



$$f(x) = x^2 * e^{-x/3}$$



$$f(x) = 1/(1 + x^2)$$



7.5 Interpretation of Results

Cubic spline interpolation builds a smooth piecewise cubic polynomial that guarantees continuous first and second derivatives, making it a powerful and stable choice for interpolating smooth functions. Across all three tested functions, **the cubic spline consistently produced accurate approximations**, with low error and no signs of divergence or instability.

7.5.1 Function 1: $f(x)=\sin(x)$, $x \in [0, 2\pi]$

- **Max Error:** ~ 0.0214
- **Remarks:** The spline interpolated the sine wave with **excellent accuracy**, maintaining phase and amplitude well across the interval. This is expected since splines are particularly effective for periodic, smooth functions over densely sampled points.

7.5.2 Function 2: $f(x)=x^2 * e^{-x/3}$, $x \in [0, 10]$

- **Max Error:** ~ 0.0460
- **Remarks:** Performance was consistently strong across the entire range. Despite the exponential term flattening the curve at larger x , the spline handled the change in curvature effectively, with errors remaining **well below 0.05** at all test points.

7.5.3 Function 3: $f(x)=1/(1 + x^2)$, $x \in [-5, 5]$

- **Max Error:** ~ 0.0118
- **Remarks:** The interpolation tracked the rational function very closely, including the sharp curvature near the origin and the flatter ends of the interval. Even at points not aligned with original nodes, **errors stayed extremely small**, demonstrating spline robustness.

7.5.4 Conclusion

Cubic spline interpolation is **superior in numerical stability and local accuracy** compared to global polynomial interpolation techniques like Hermite or Lagrange. Key strengths:

- **No overshooting or oscillations** (unlike high-degree polynomials),
- **High smoothness** (C^2 continuity),
- **Consistent precision** across diverse function types.

8 Parametric Curves

8.1 Method Overview

- Purpose: To represent smooth curves using control points, allowing flexible manipulation of the curve's shape.
- Based on a polynomial function that defines a curve in terms of control points, where the curve is influenced by the positions of the control points.
- Steps:
 - Given $n+1$ control points P_0, P_1, \dots, P_n .
 - The Bézier curve is defined as:
 - $B(t) = \sum_{i=0}^n \binom{n}{i} (1-t)^{n-i} t^i P_i, \quad t \in [0,1]$
 - The curve is a weighted sum of the control points, where the weights are determined by the Bernstein polynomials.
- Convergence: The curve always passes through the first and last control points P_0 and P_n .

8.2 Explanation of Code

8.2.1 evaluateBezier Function

- **Purpose:** Computes a point on a cubic Bézier curve at a specific parameter t using equations (3.25) and (3.26) from the book.
- **Inputs:**
 - $p0, p1$: Endpoints of the curve
 - $guide0, guide1$: Guide points used to derive control values (α, β)
 - t : A parameter between 0 and 1
- **Output:** Returns a point (x, y) on the Bézier curve

8.2.2 plotBezierCurve Function

- **Purpose:** Visualizes the cubic Bézier curve and outputs numerical data
- **Operations:**
 - Displays a table of input parameters (endpoints and guide points)
 - Evaluates the Bézier curve at multiple t values and prints the resulting points in a table
 - Plots the curve, endpoints, guide lines, and guide points using matplotlib

8.2.3 main Function

- **Purpose:** Provides an example case for visualizing a cubic Bézier curve

- **Example Setup:**
 - Uses specific values from a textbook example (Figure 3.18)
 - Calls the plotting function and displays the plot
 - Handles and re-raises exceptions for detailed error reporting

8.3 Graph and Data

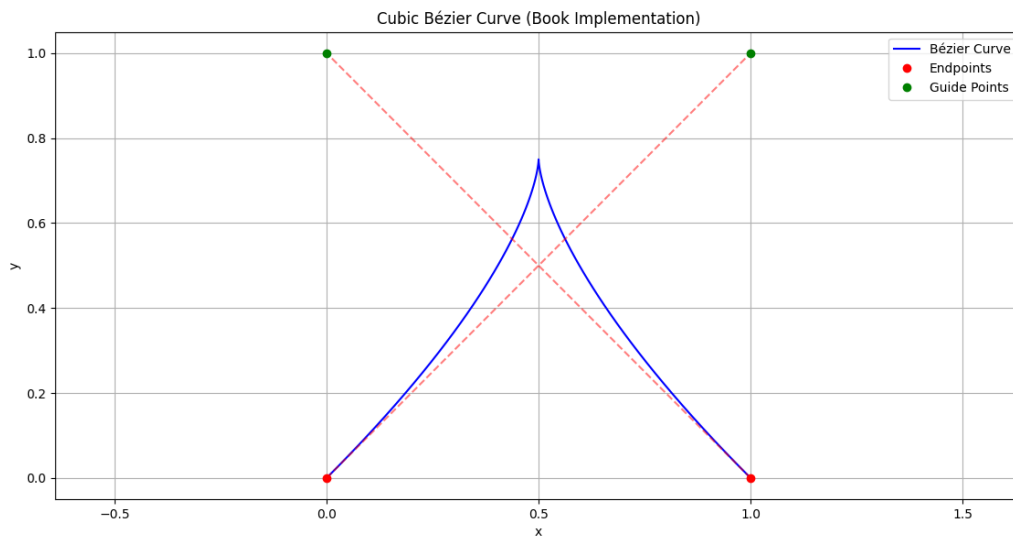
8.3.1 Parametric Curve 1

Control Points

Point Type	X	Y
Endpoint 1	0.0	0.0
Guide Point 1	1.0	1.0
Guide Point 2	0.0	1.0
Endpoint 2	1.0	0.0

Points on the Bézier Curve

T	x(t)	y(t)
0.0000	0.0000	0.0000
0.0526	0.1419	0.1496
0.1053	0.2540	0.2825
0.1579	0.3398	0.3989
0.2105	0.4030	0.4986
0.2632	0.4469	0.5817
0.3158	0.4750	0.6482
0.3684	0.4909	0.6981
0.4211	0.4980	0.7313
0.4737	0.4999	0.7479
0.5263	0.5001	0.7479
0.5789	0.5020	0.7313
0.6316	0.5091	0.6981
0.6842	0.5250	0.6482
0.7368	0.5531	0.5817
0.7895	0.5970	0.4986
0.8421	0.6602	0.3989
0.8947	0.7460	0.2825
0.9474	0.8581	0.1496
1.0000	1.0000	0.0000



8.3.2 Parametric Curve 2

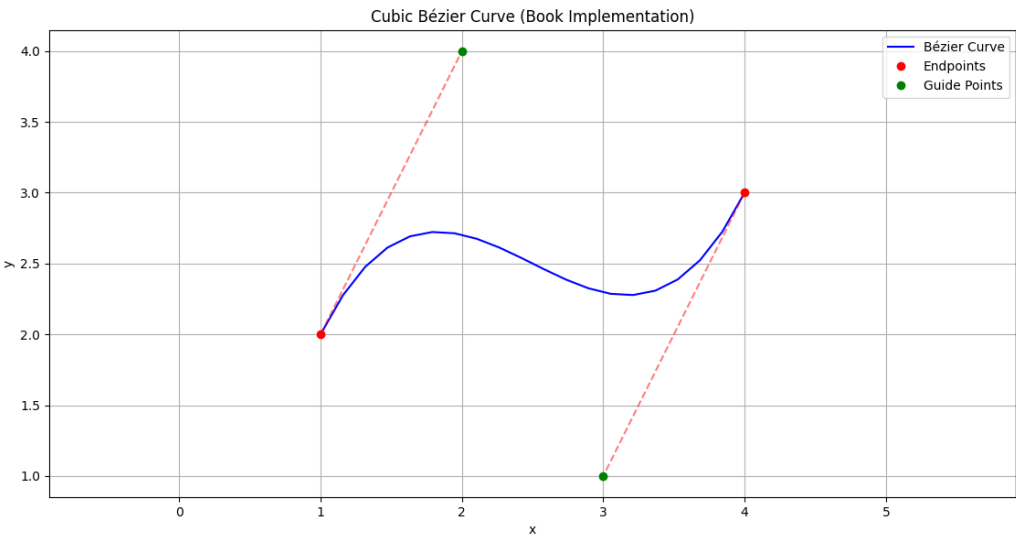
Control Points

Point Type	X	Y
Endpoint 1	1.0	2.0
Guide Point 1	2.0	4.0
Guide Point 2	3.0	1.0
Endpoint 2	4.0	3.0

Points on the Bézier Curve

t	x(t)	y(t)
0.0000	1.0000	2.0000
0.0526	1.1579	2.2757
0.1053	1.3158	2.4770
0.1579	1.4737	2.6128
0.2105	1.6316	2.6916
0.2632	1.7895	2.7224
0.3158	1.9474	2.7138
0.3684	2.1053	2.6746
0.4211	2.2632	2.6135
0.4737	2.4211	2.5393
0.5263	2.5789	2.4607

t	x(t)	y(t)
0.5789	2.7368	2.3865
0.6316	2.8947	2.3254
0.6842	3.0526	2.2862
0.7368	3.2105	2.2776
0.7895	3.3684	2.3084
0.8421	3.5263	2.3872
0.8947	3.6842	2.5230
0.9474	3.8421	2.7243
1.0000	4.0000	3.0000



8.3.3 Parametric Curve 3

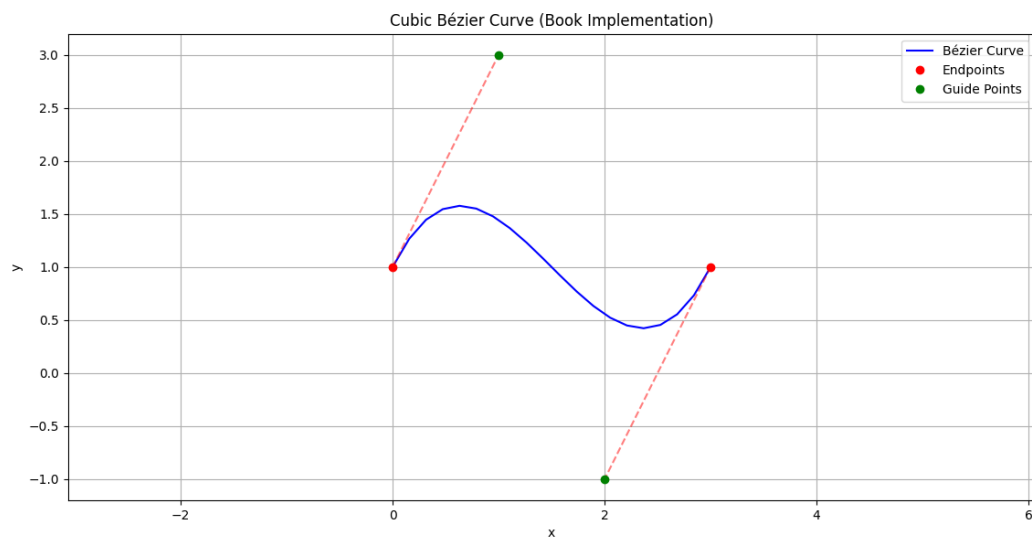
Control Points

Point Type	X	Y
Endpoint 1	0.0	1.0
Guide Point 1	1.0	3.0
Guide Point 2	2.0	-1.0
Endpoint 2	3.0	1.0

Points on the Bézier Curve

t	x(t)	y(t)
0.0000	0.0000	1.0000

t	x(t)	y(t)
0.0526	0.1579	1.2677
0.1053	0.3158	1.4461
0.1579	0.4737	1.5459
0.2105	0.6316	1.5773
0.2632	0.7895	1.5511
0.3158	0.9474	1.4776
0.3684	1.1053	1.3674
0.4211	1.2632	1.2309
0.4737	1.4211	1.0787
0.5263	1.5789	0.9213
0.5789	1.7368	0.7691
0.6316	1.8947	0.6326
0.6842	2.0526	0.5224
0.7368	2.2105	0.4489
0.7895	2.3684	0.4227
0.8421	2.5263	0.4541
0.8947	2.6842	0.5539
0.9474	2.8421	0.7323
1.0000	3.0000	1.0000



9 Numerical Differentiation

9.1 Method Overview

- Purpose: To approximate the derivative of a function at a given point using discrete data points.
- Based on using finite differences to estimate the rate of change of the function.
- Steps:
 - For forward difference:
 - $f'(x) \approx \frac{f(x+h)-f(x)}{h}$
 - For backward difference:
 - $f'(x) \approx \frac{f(x)-f(x-h)}{h}$
 - For central difference:
 - $f'(x) \approx \frac{f(x+h)-f(x-h)}{2h}$
 - Choose h(step size) small enough for accuracy, but not so small that it introduces numerical errors.
- Convergence: The accuracy increases as h decreases, but numerical errors may grow if h is too small.
- Advantages: Simple and fast, easy to apply when only discrete data is available.
- Disadvantages: Sensitive to choice of h, may introduce significant errors for poorly chosen step sizes.

9.2 Explanation of Code

forwardDifference / backwardDifference / centralDifference:

- These functions approximate the first derivative using basic finite difference formulas
- Forward and backward use adjacent points, while central uses symmetric points for better accuracy

threePointEndpoint / fivePointMidpoint:

- These use more points to achieve higher accuracy
- The three-point endpoint is useful at boundaries; the five-point midpoint provides a very accurate central estimate

compareMethods:

- Computes numerical derivatives using all methods at a specific point
- Compares results with the analytical derivative and calculates absolute and relative errors
- Returns a formatted DataFrame to summarize performance

plotApproximations:

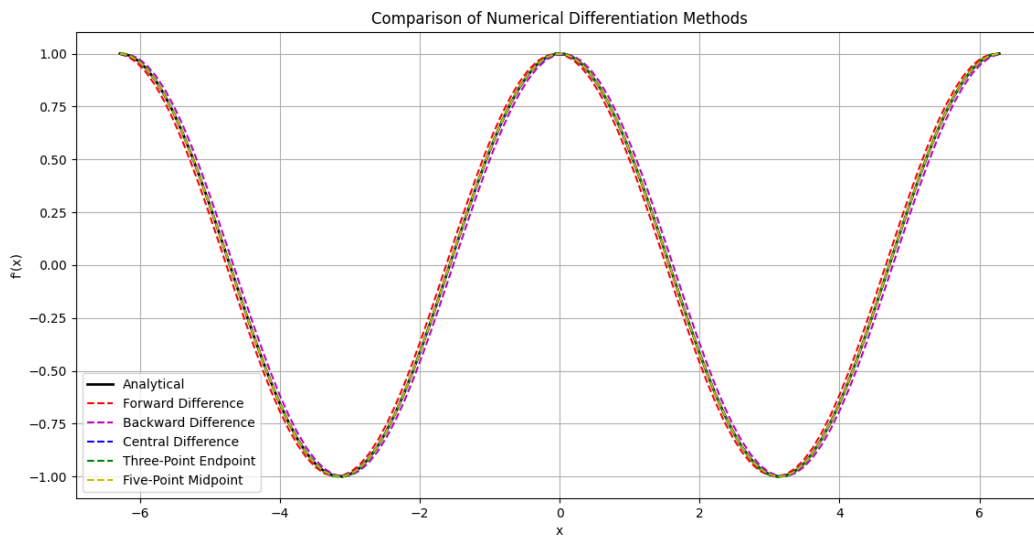
- Visualizes numerical derivatives and the analytical derivative across a range
- Shows how each method behaves over a continuous interval

9.3 Results and Graphs

$$f(x) = \sin(x), f'(x) = \cos(x)$$

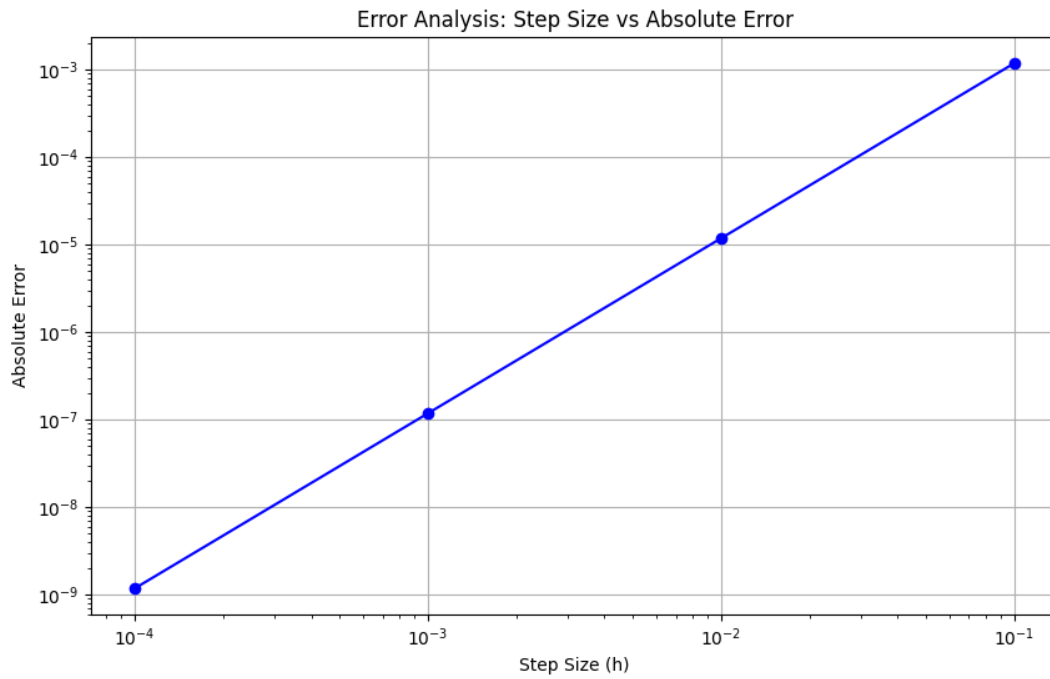
Approximating around $x = 0.7854$ and $h = 0.1$

Method	Approximation	Absolute Error	Relative Error
Forward Difference	0.6706029729	0.0365038083	0.0516241808
Backward Difference	0.7412547451	0.0341479639	0.0482925137
Central Difference	0.7059288590	0.0011779222	0.0016658335
Three-Point Endpoint	0.7092790806	0.0021722994	0.0030720953
Five-Point Midpoint	0.7071044270	0.0000023542	0.0000033294



Effect of Step Size on Central Difference Method

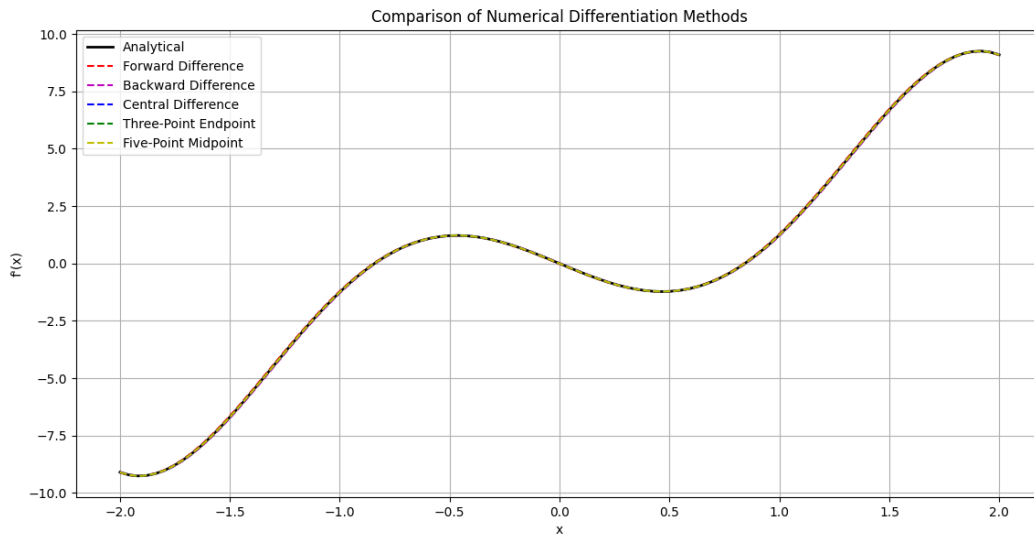
Step Size (h)	Approximation	True Value	Absolute Error
0.1000000000	0.7059288590	0.7071067812	0.0011779222
0.0100000000	0.7070949961	0.7071067812	0.0000117851
0.0010000000	0.7071066633	0.7071067812	0.0000001179
0.0001000000	0.7071067800	0.7071067812	0.0000000012



$$f(x) = x^3 * \sin(x) + \cos(2x), f'(x) = 3x^2 * \sin(x) + x^3 * \cos(x) - 2\sin(2x)$$

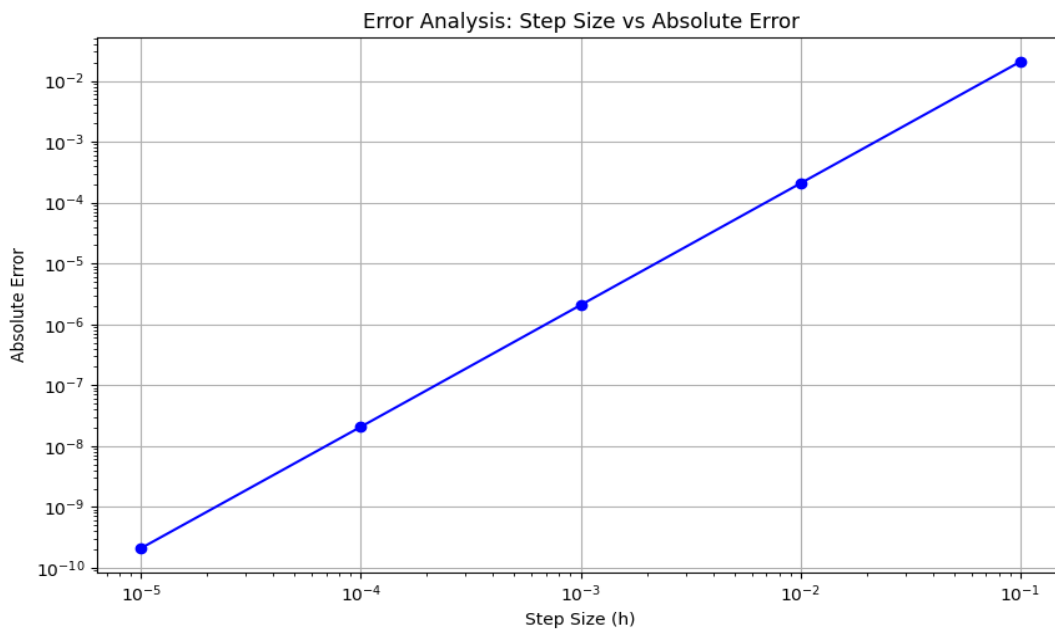
Approximating around $x = 1.0472$ and $h = 0.01$

Method	Approximation	Absolute Error	Relative Error
Forward Difference	1.7401382699	0.0488893897	0.0289072710
Backward Difference	1.6427737283	0.0484751519	0.0286623409
Central Difference	1.6914559991	0.0002071189	0.0001224651
Three-Point Endpoint	1.6908432968	0.0004055834	0.0002398130
Five-Point Midpoint	1.6912489123	0.0000000321	0.0000000190



Effect of Step Size on Central Difference Method

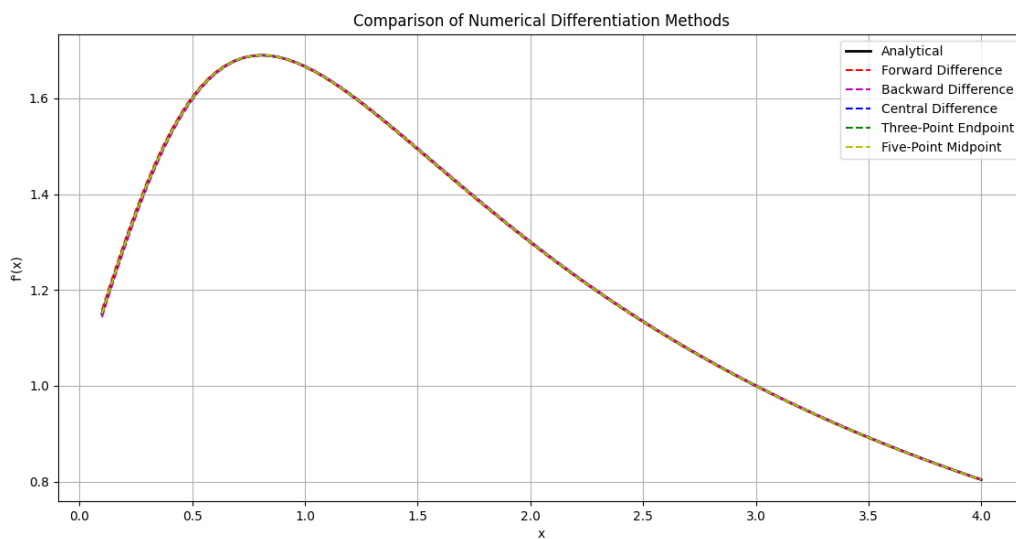
Step Size (h)	Approximation	True Value	Absolute Error
0.1000000000	1.7118814138	1.6912488802	0.0206325336
0.0100000000	1.6914559991	1.6912488802	0.0002071189
0.0010000000	1.6912509515	1.6912488802	0.0000020713
0.0001000000	1.6912489009	1.6912488802	0.0000000207
0.0000100000	1.6912488804	1.6912488802	0.0000000002



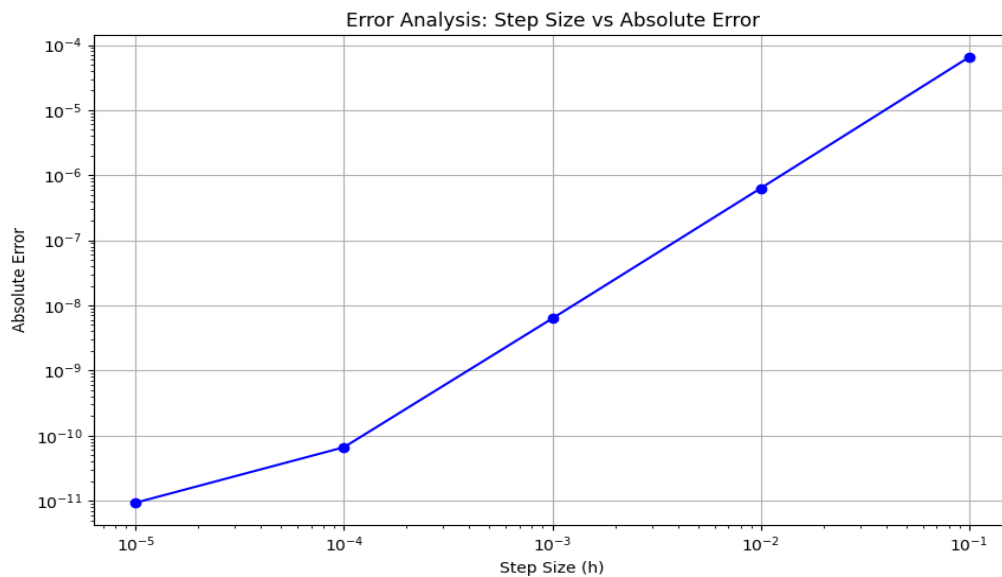
$$f(x) = \ln(x^2 + 1) + 2\ln(x + 2), f'(x) = (2x)/(x^2 + 1) + 2/(x + 2)$$

Approximating around $x = 1.5$ and $h = 0.01$

Method	Approximation	Absolute Error	Relative Error
Forward Difference	1.4925051359	0.0020003586	0.0013384752
Backward Difference	1.4965045929	0.0019990984	0.0013376320
Central Difference	1.4945048644	0.0000006301	0.0000004216
Three-Point Endpoint	1.4945065770	0.0000010825	0.0000007243
Five-Point Midpoint	1.4945054953	0.0000000008	0.0000000005



Effect of Step Size on Central Difference Method



9.4 Interpretation of Results

Numerical differentiation offers effective approximations of derivatives when analytic forms are hard to compute or unavailable (e.g., experimental data). This analysis compares various differentiation methods across three functions, showing how accuracy and precision evolve with method complexity and step size.

9.4.1 $f(x)=\sin(x)$, $f'(x)=\cos(x)$ at $x \approx \frac{\pi}{4}$

- **True Derivative:** ~ 0.7071
- **Best Method:** Five-Point Midpoint (Error $\approx 2.35e-6$)
- **Observation:**
 - Central and five-point methods are **extremely close to the true value**.
 - As h decreases, error drops sharply, reaching **machine precision** at $h = 0.0001$.
 - Simpler methods like forward/backward differences show noticeable error (~ 0.036 – 0.034).

9.4.2 $f(x) = x^3 \sin(x) + \cos(2x)$

- **True Derivative at $x=1.0472$:** ~ 1.69125
- **Best Method:** Five-Point Midpoint (Error $\approx 3.21e-8$)
- **Observation:**
 - **All methods improve with smaller step size**, especially central and five-point.
 - Forward/backward differences show larger deviation (~ 0.049 absolute error).
 - This confirms that **higher-order methods are much more effective** for complex expressions.

9.4.3 $f(x) = \ln(x^2 + 1) + 2 \ln(x + 2)$

- **True Derivative at $x=1.5$:** ~ 1.494505
- **Best Method:** Five-Point Midpoint (Error $\approx 8e-10$)
- **Observation:**
 - Even the basic central difference is **accurate to 6 decimal places**.
 - The five-point method achieves **almost perfect agreement** with the analytic derivative.
 - All methods perform better here due to the function's smoothness.

9.5 Conclusion

- **Best Overall Method:**
The **Five-Point Midpoint** method is the most accurate across all tests, especially with a sufficiently small h .
- **Step Size Matters:**
Reducing h significantly enhances accuracy—but only up to a point. Extremely small h may cause **floating-point round-off errors**.

10 Richardson Extrapolation

10.1 Method Overview

- Purpose: To improve the accuracy of numerical differentiation by eliminating leading-order error terms.
- Based on using two different step sizes to extrapolate and remove the error, yielding a more accurate approximation of the derivative.
- Steps:
 - Use two finite difference approximations with different step sizes h and $h/2$:
 - $D(h) = \frac{f(x+h)-f(x)}{h}$, $D(h/2) = \frac{f(x+h/2)-f(x)}{h/2}$
 - Combine the two estimates using:
 - $D_{\text{extrapolated}} = \frac{4D(h/2)-D(h)}{3}$
- Convergence: Provides higher-order accuracy by eliminating leading errors, improving accuracy by a factor of 4.
- Advantages: Increases the accuracy of derivative approximations without requiring smaller step sizes.
- Disadvantages: Requires two evaluations of the function at different step sizes, leading to higher computational cost.

10.2 Explanation of Code

forwardDifference:

- Approximates the first derivative using the basic forward difference formula
- Used as the base method for Richardson refinement

richardsonExtrapolation:

- Builds a triangular extrapolation table D using forward difference at decreasing step sizes $h / 2^i$
- Applies Richardson's formula to iteratively improve derivative accuracy
- Returns the most accurate estimate from the top of the table

computeRichardsonTable:

- Constructs the full extrapolation table for display
- Each entry $D[i, j]$ combines previous estimates to remove more error terms
- Outputs results as a well-formatted DataFrame with labeled rows and columns

plotLevelVsError:

- Plots the absolute error (in log scale) against the extrapolation level
- Visually demonstrates the convergence and accuracy improvement with increasing levels
- Annotates each point with its error value for clarity

10.3 Table and Graphs

$$f(x) = \sin(x), f'(x) = \cos(x)$$

Approximating around $x = 0.7854$

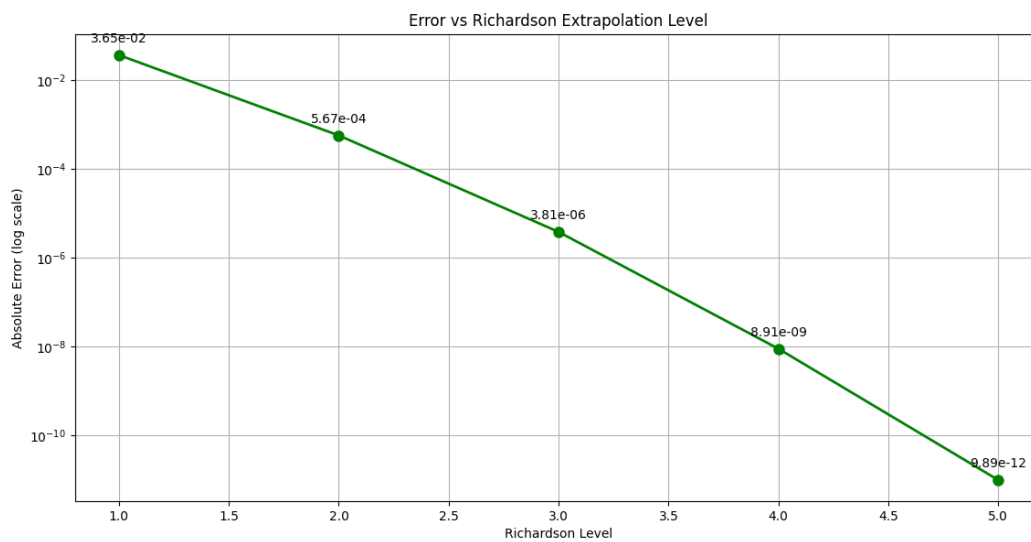
Richardson Extrapolation Table

Step Size	$O(h^1)$	$O(h^2)$	$O(h^3)$	$O(h^4)$	$O(h^5)$
$h/1$	0.6706029729	0.7076734335	0.7071105902	0.7071067723	0.7071067812
$h/2$	0.6891382032	0.7072513010	0.7071072495	0.7071067806	0.0000000000
$h/4$	0.6981947521	0.7071432624	0.7071068392	0.0000000000	0.0000000000
$h/8$	0.7026690073	0.7071159450	0.0000000000	0.0000000000	0.0000000000
$h/16$	0.7048924761	0.0000000000	0.0000000000	0.0000000000	0.0000000000

True Derivative at $x = 0.7854$: 0.7071067812

Improvement with Increasing Richardson Levels

Level	Approximation	Error
1	0.6706029729	3.6503808283e-02
2	0.7076734335	5.6665230667e-04
3	0.7071105902	3.8090201756e-06
4	0.7071067723	8.9141927173e-09
5	0.7071067812	9.8872021681e-12



$$f(x) = x^3 * \sin(x) + \cos(2x), f'(x) = 3x^2 * \sin(x) + x^3 * \cos(x) - 2\sin(2x)$$

Approximating around $x = 1.0472$

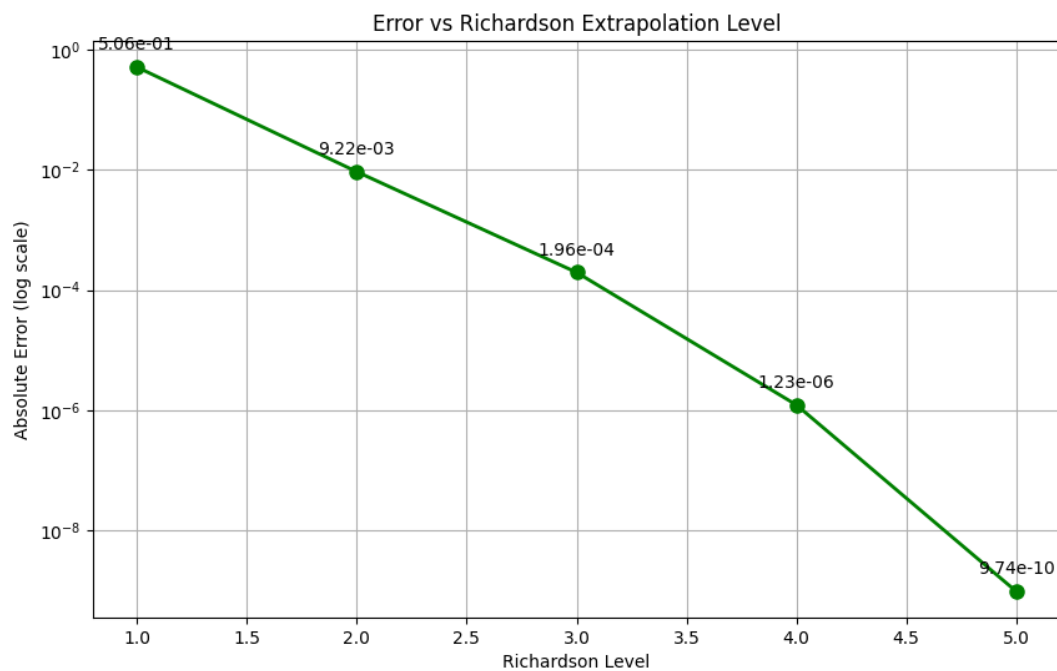
Richardson Extrapolation Table

Step Size	$O(h^1)$	$O(h^2)$	$O(h^3)$	$O(h^4)$	$O(h^5)$
h/1	2.1972928535	1.6820316530	1.6910532885	1.6912501058	1.6912488812
h/2	1.9396622532	1.6887978796	1.6912255036	1.6912489577	0.0000000000
h/4	1.8142300664	1.6906185976	1.6912460260	0.0000000000	0.0000000000
h/8	1.7524243320	1.6910891689	0.0000000000	0.0000000000	0.0000000000
h/16	1.7217567504	0.0000000000	0.0000000000	0.0000000000	0.0000000000

True Derivative at $x = 1.0472$: 1.6912488802

Improvement with Increasing Richardson Levels

Level	Approximation	Error
1	2.1972928535	5.0604397326e-01
2	1.6820316530	9.2172271972e-03
3	1.6910532885	1.9559175463e-04
4	1.6912501058	1.2255680022e-06
5	1.6912488812	9.7382235609e-10



$$f(x) = \ln(x^2 + 1) + 2\ln(x + 2), f'(x) = (2x)/(x^2 + 1) + 2/(x + 2)$$

Approximating around $x = 1.5$

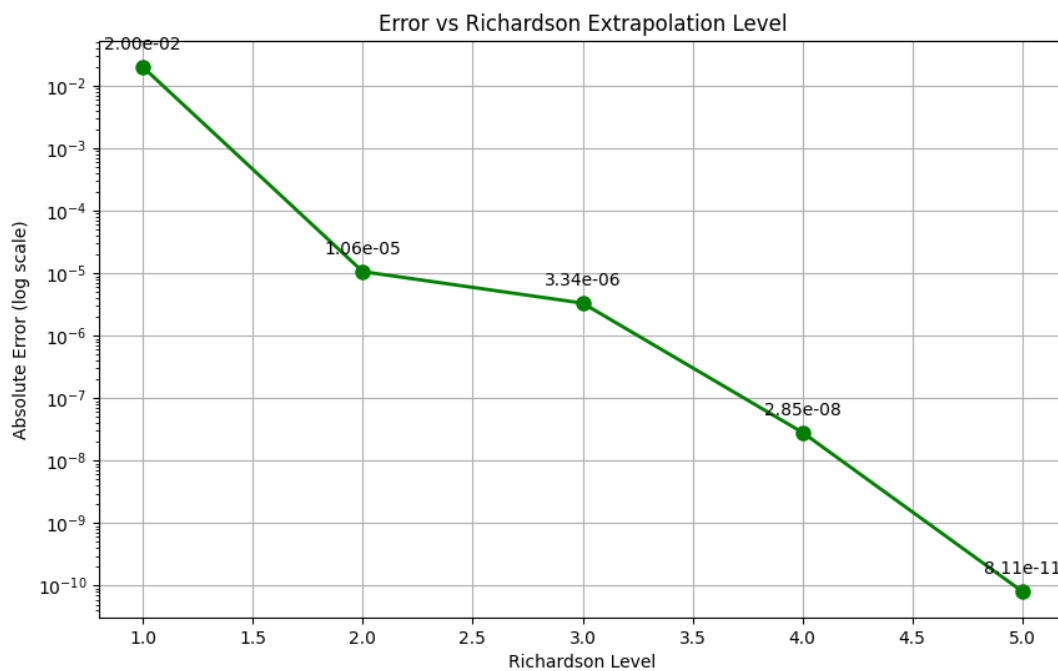
Richardson Extrapolation Table

Step Size	$O(h^1)$	$O(h^2)$	$O(h^3)$	$O(h^4)$	$O(h^5)$
$h/1$	1.4744730246	1.4945161429	1.4945088343	1.4945055230	1.4945054946
$h/2$	1.4844945837	1.4945106615	1.4945059369	1.4945054964	0.0000000000
$h/4$	1.4895026226	1.4945071180	1.4945055514	0.0000000000	0.0000000000
$h/8$	1.4920048703	1.4945059431	0.0000000000	0.0000000000	0.0000000000
$h/16$	1.4932554067	0.0000000000	0.0000000000	0.0000000000	0.0000000000

True Derivative at $x = 1.5$: 1.4945054945

Improvement with Increasing Richardson Levels

Level	Approximation	Error
1	1.4744730246	2.0032469949e-02
2	1.4945161429	1.0648367676e-05
3	1.4945088343	3.3398050909e-06
4	1.4945055230	2.8482136871e-08
5	1.4945054946	8.1067375035e-11



10.4 Interpretation of Results

Richardson Extrapolation significantly enhances the accuracy of numerical derivatives by combining estimates at progressively smaller step sizes. This method systematically **removes lower-order error terms**, producing results that converge rapidly to the true derivative—even from a basic forward difference starting point.

10.4.1 $f(x)=\sin(x)$, $f'(x)=\cos(x)$ at $x \approx \frac{\pi}{4}$

- **True Derivative:** ~ 0.7071067812
- **Best Estimate:** Level 5 \rightarrow Error $\approx 9.89\text{e-}12$
- **Observation:**
 - Rapid convergence from level 1 (error $\sim 3.65\text{e-}2$) to level 5 (error $\sim 1\text{e-}11$).
 - At level 3 onward, error becomes **nearly negligible**, reaching **machine precision** by level 5.

10.4.2 $f(x) = x^3 \sin(x) + \cos(2x)$

- **True Derivative at $x=1.0472x = 1.0472$:** ~ 1.6912488802
- **Best Estimate:** Level 5 \rightarrow Error $\approx 9.74\text{e-}10$
- **Observation:**
 - Starts with a poor approximation (error ~ 0.506 at level 1), but converges rapidly.
 - Level 3 already gives a **close estimate**, and level 5 reaches **sub-nano level precision**.

10.4.3 $f(x) = \ln(x^2 + 1) + 2 \ln(x + 2)$

- **True Derivative at $x=1.5x = 1.5$:** ~ 1.4945054945
- **Best Estimate:** Level 5 \rightarrow Error $\approx 8.11\text{e-}11$
- **Observation:**
 - All intermediate estimates approach the exact value closely.
 - The refinement from level 2 onwards is already **very precise**.
 - Highlights how well the method performs for **logarithmic functions** with smooth curvature.

Conclusion

- **Accuracy Enhancement:**
Richardson Extrapolation yields **multiple orders of magnitude improvement** in derivative accuracy, even when starting from the crude forward difference method.
- **Convergence Efficiency:**
Typically, **5 levels are sufficient** to reach near machine precision, assuming smooth functions and accurate arithmetic.

11 Numerical Integration

11.1 Method Overview

- Purpose: To approximate the integral of a function over a given interval using discrete data points or methods.
- Based on summing up areas under curves using different approximation techniques.
- Steps:
 - For **Rectangular (Left Riemann) Method**:
 - $I \approx \sum_{i=0}^{n-1} f(x_i) \cdot \Delta$
 - For **Trapezoidal Rule**:
 - $I \approx \frac{\Delta x}{2} [f(x_0) + 2 \sum_{i=1}^{n-1} f(x_i) + f(x_n)]$
 - For **Simpson's Rule**:
 - $I \approx \frac{\Delta x}{3} [f(x_0) + 4 \sum_{i=1,3,5,\dots} f(x_i) + 2 \sum_{i=2,4,6,\dots} f(x_i) + f(x_n)]$
- Convergence: Accuracy increases as the step size Δx decreases, but computational cost increases.
- Advantages: Simple to implement, effective for functions that are smooth or can be approximated by polynomials.
- Disadvantages: Can be inaccurate for complex or oscillatory functions, requires fine step sizes for higher accuracy.

11.2 Explanation of Code

trapezoidalRule:

- Divides the interval $[a,b]$ into n subintervals of equal width
- Applies the trapezoidal formula using the average height of consecutive function values
- Returns the area estimate as a weighted sum of function evaluations

simpsonsRule:

- Divides the interval into an even number of subintervals
- Applies Simpson's 1/3 Rule: uses function values at even and odd indices with different weights (4 and 2)
- Returns a more accurate area estimate due to parabolic interpolation

analyzeConvergence:

- Evaluates both integration methods for increasing $n = 2^i$ (refinement levels)
- Compares each result to a high-precision "exact" value to compute absolute error
- Returns a table showing convergence trends as subintervals increase

plotErrorConvergence:

- Plots the logarithmic error of both methods against the number of subintervals
- Visualizes convergence behavior and shows how Simpson's Rule achieves better accuracy for the same n
- Helps assess method efficiency graphically

visualizeIntegration:

- Plots the function over [a, b] alongside graphical representations of both integration methods
- Shows trapezoids under the curve and function samples used in Simpson's method
- Enhances understanding by combining visual cues with function sampling

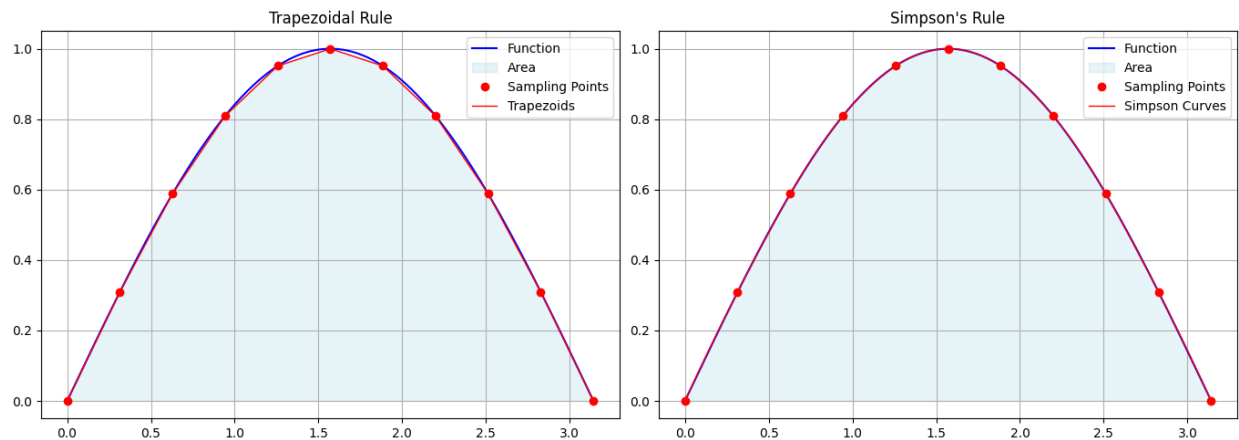
11.3 Results and Graphs

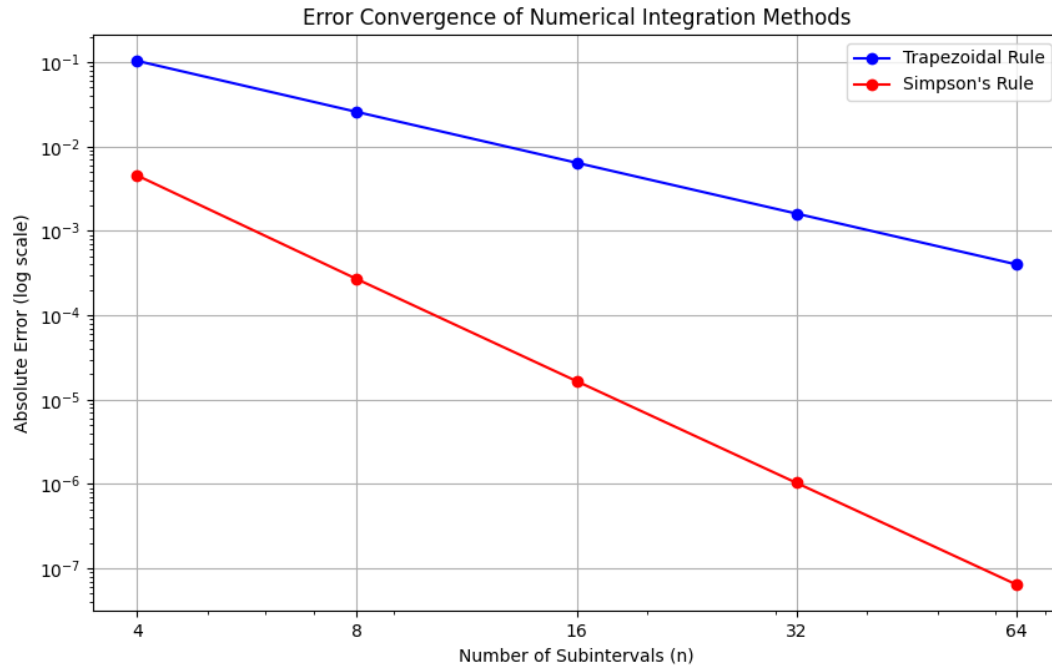
$$f(x) = \sin(x)$$

11.3.1.1 Results with 10 Subintervals:

- **Trapezoidal Rule:** 1.9835235375
- **Simpson's Rule:** 2.0001095173
- **Exact Value:** 2.0000000000

Subintervals	Trapezoidal Error	Simpson Error
4	1.0388110206e-01	4.5597549844e-03
8	2.5768398054e-02	2.6916994839e-04
16	6.4296562277e-03	1.6591047935e-05
32	1.6066390299e-03	1.0333694127e-06
64	4.0161135996e-04	6.4530001787e-08



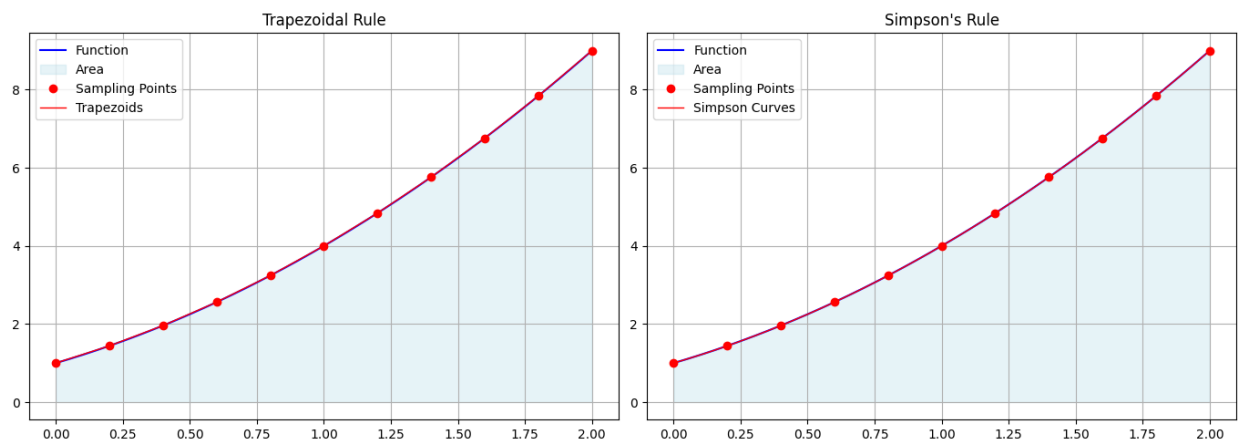


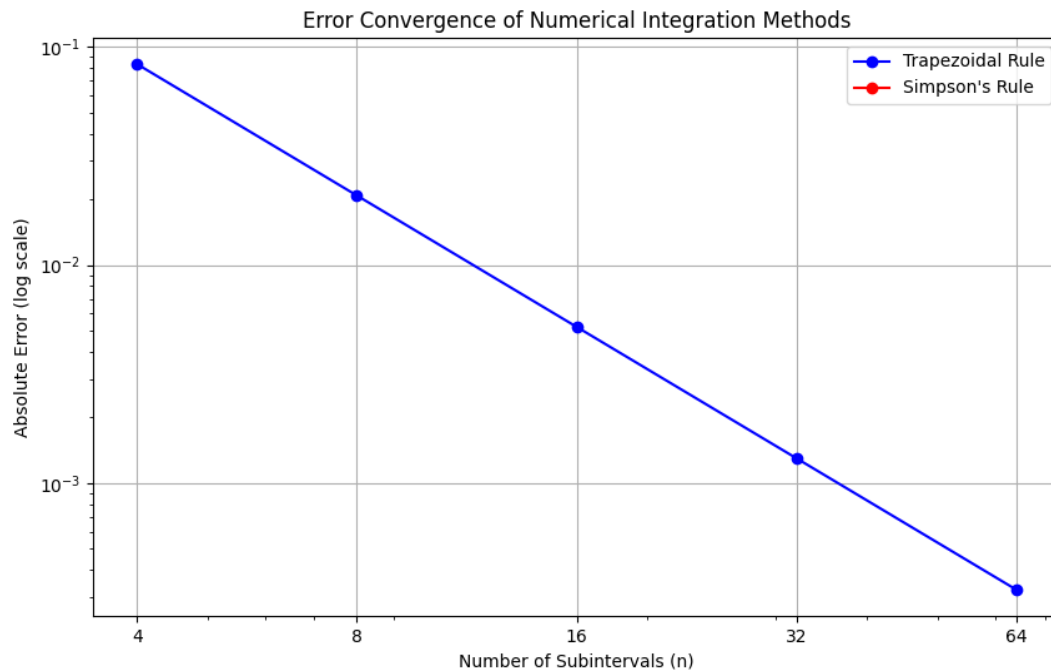
$$f(x) = x^2 + 2x + 1$$

11.3.1.2 Results with 10 Subintervals:

- **Trapezoidal Rule:** 8.6800000000
- **Simpson's Rule:** 8.6666666667
- **Exact Value:** 8.6666666667

Subintervals	Trapezoidal Error	Simpson Error
4	8.333333333e-02	0.000000000e+00
8	2.083333333e-02	0.000000000e+00
16	5.208333333e-03	0.000000000e+00
32	1.302083333e-03	0.000000000e+00
64	3.252083333e-04	0.000000000e+00



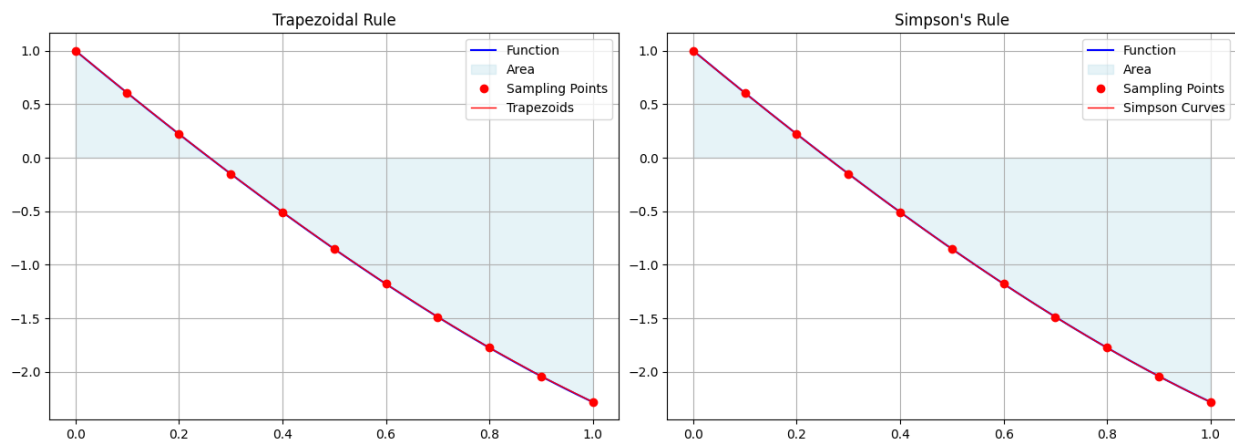


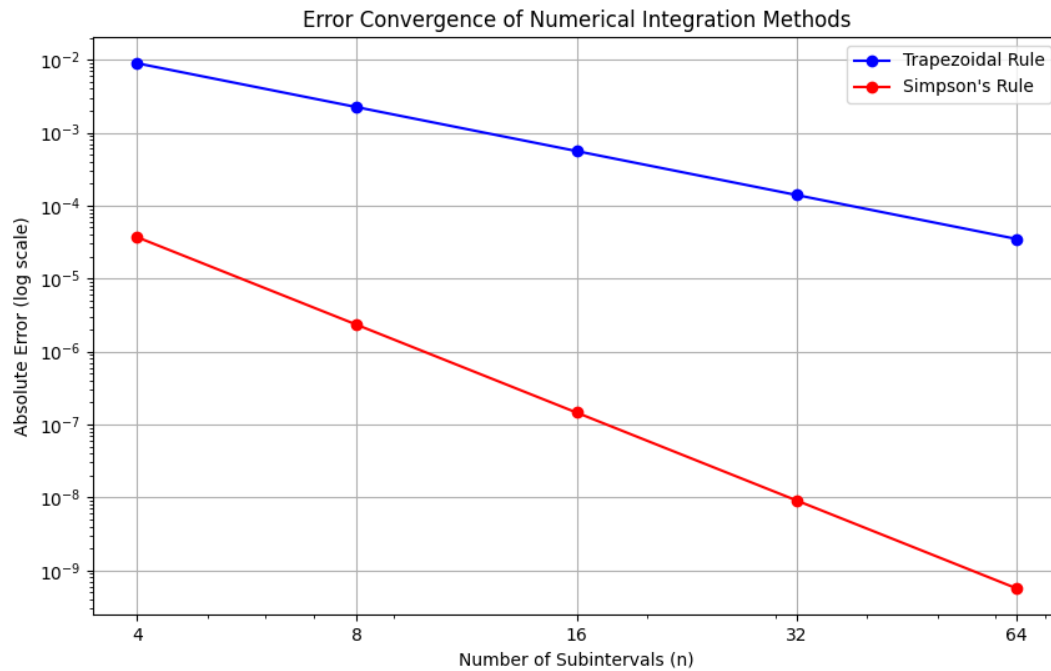
$$f(x) = e^x - 5x$$

11.3.1.3 Results with 10 Subintervals:

- **Trapezoidal Rule:** -0.7802865086
- **Simpson's Rule:** -0.7817172181
- **Exact Value:** -0.7817181715

Subintervals	Trapezoidal Error	Simpson Error
4	8.9400760985e-03	3.7013462702e-05
8	2.2367637053e-03	2.3262408518e-06
16	5.5930012095e-04	1.4559284689e-07
32	1.3983185728e-04	9.1027265725e-09
64	3.4958391048e-05	5.6897020428e-10





11.4 Interpretation of Results

Numerical integration methods are designed to approximate the area under a curve when an analytical solution is difficult or impossible to obtain. The **Trapezoidal Rule** approximates the integral by dividing the area into trapezoids, while **Simpson's Rule** uses parabolic segments to achieve higher accuracy. As the number of subintervals increases, the accuracy of both methods improves, with Simpson's Rule showing superior performance for the same number of subintervals.

11.4.1 $f(x)=\sin(x)$

- **Exact Value:** 2.0000000000
- **Results with 10 Subintervals:**
 - **Trapezoidal Rule:** 1.9835235375
 - **Simpson's Rule:** 2.0001095173
- **Error Convergence:**
 - **Trapezoidal Rule:** Error decreases as the number of subintervals increases, reaching **$4.02e-4$** at 64 subintervals.
 - **Simpson's Rule:** Error is significantly smaller, decreasing to **$6.45e-8$** at 64 subintervals.

Observation:

- **Simpson's Rule** outperforms Trapezoidal, providing a more accurate estimate even with fewer subintervals. The accuracy improvement is evident as the error falls much faster.

11.4.2 $f(x)=x^2 + 2x + 1$

- **Exact Value:** 8.6666666667
- **Results with 10 Subintervals:**
 - **Trapezoidal Rule:** 8.6800000000
 - **Simpson's Rule:** 8.6666666667
- **Error Convergence:**
 - **Trapezoidal Rule:** The error remains around **8.33e-2** for 4 subintervals, but decreases significantly as subintervals increase, reaching **3.26e-4** at 64 subintervals.
 - **Simpson's Rule:** No error at higher subintervals, consistently producing the exact value.

Observation:

- **Simpson's Rule** delivers the exact result right from the first refinement level (subintervals = 8), while the Trapezoidal Rule requires more refinement to approach the exact value.

11.4.3 $f(x) = e^x - 5x$

- **Exact Value:** -0.7817181715
- **Results with 10 Subintervals:**
 - **Trapezoidal Rule:** -0.7802865086
 - **Simpson's Rule:** -0.7817172181
- **Error Convergence:**
 - **Trapezoidal Rule:** The error decreases from **8.94e-3** at 4 subintervals to **3.50e-5** at 64 subintervals.
 - **Simpson's Rule:** The error decreases from **3.70e-5** at 4 subintervals to **5.69e-10** at 64 subintervals.

Observation:

- **Simpson's Rule** shows superior accuracy, even with fewer subintervals, achieving very small errors by level 8 and staying within **machine precision** at level 64.

11.4.4 Conclusion

- **Trapezoidal Rule:**
 - Provides **reasonable accuracy** for functions that are approximately linear over small intervals.
 - Converges **linearly** as subintervals increase.
- **Simpson's Rule:**
 - Provides **higher accuracy** due to parabolic interpolation.
 - Converges **quadratically**, making it a better choice for smoother functions or when precision is critical.

12 Romberg Integration

12.1 Method Overview

- Purpose: To improve the accuracy of numerical integration by applying Richardson extrapolation to the trapezoidal rule.
- Based on recursively refining trapezoidal approximations and eliminating error terms using extrapolation.
- Steps:
 - Compute trapezoidal approximations: $T(h), T(h/2), T(h/4), \dots$
 - Apply Richardson extrapolation using:
 - $R(k, j) = R(k, j - 1) + \frac{1}{4^{j-1} - 1} (R(k, j - 1) - R(k - 1, j - 1))$
 - Continue until the difference between successive estimates is within a specified tolerance.
- Convergence: Rapid convergence for smooth functions; accuracy increases exponentially with each level.
- Advantages: Very accurate for smooth integrands, systematic refinement without decreasing step size drastically.
- Disadvantages: Computationally intensive, less effective for functions with discontinuities or singularities.

12.2 Explanation of Code

trapezoidalRule:

This function computes the trapezoidal rule approximation for a given function over the interval [a, b] using n subintervals. The formula calculates the area by averaging the function values at the endpoints and summing the function values at intermediate points, weighted appropriately by the step size h.

rombergIntegration:

This function builds the **Romberg table** for a function ff over the interval [a, b] with a specified maximum refinement level (maxLevel). Initially, the first column of the table is filled using the trapezoidal rule with increasing numbers of subintervals (powers of 2). Then, the table is refined using Richardson extrapolation, where each entry $R(i, j)$ is computed from previous values in the table, utilizing the formula:

$$R(i, j) == \frac{4^j R(i + 1, j - 1) - R(i, j - 1)}{4^j - 1}$$

This extrapolation significantly improves the accuracy of the integral estimate.

displayRombergTable:

This function takes the Romberg table R and displays it in a readable format using pandas, labeling the rows and columns according to the refinement level and number of subintervals. The output is a table where each entry represents the integral estimate at a particular level of refinement.

plotConvergence:

This function plots the convergence of the Romberg integration method, comparing the standard error at each refinement level to the minimum error observed at each level. It uses a logarithmic scale to represent

the absolute errors, making it easier to assess how quickly the method converges to the true value as more levels are added. The plot also annotates each point with the corresponding error values.

visualizeIntegration:

This function visualizes the integration process using trapezoidal approximations, where the function `ff` is plotted over the interval `[a, b]` along with the trapezoids used to estimate the area. It provides a graphical view of the approximation process for better understanding.

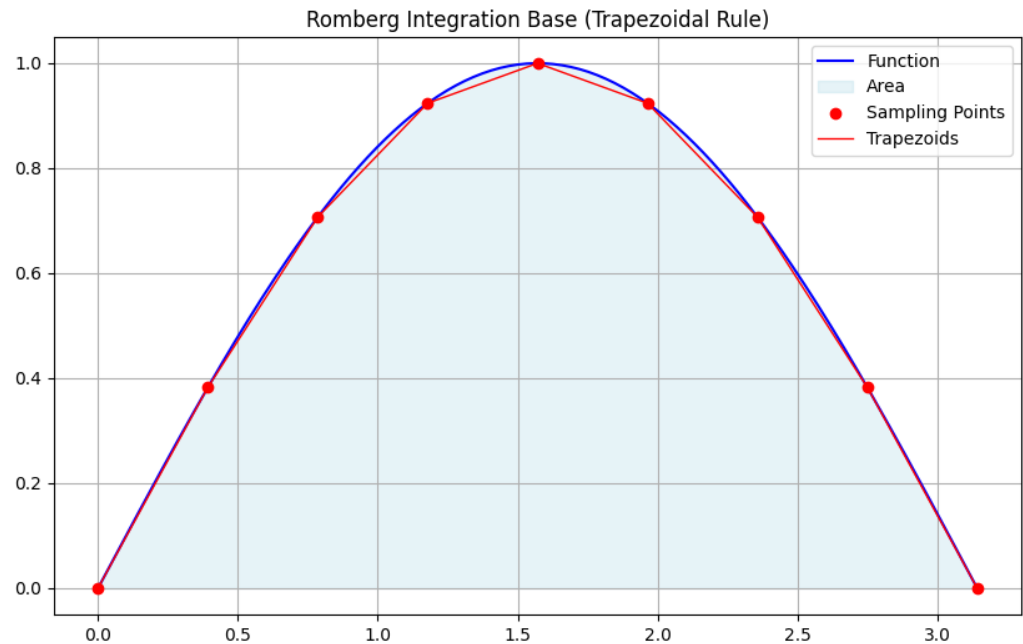
12.3 Results and Graphs:

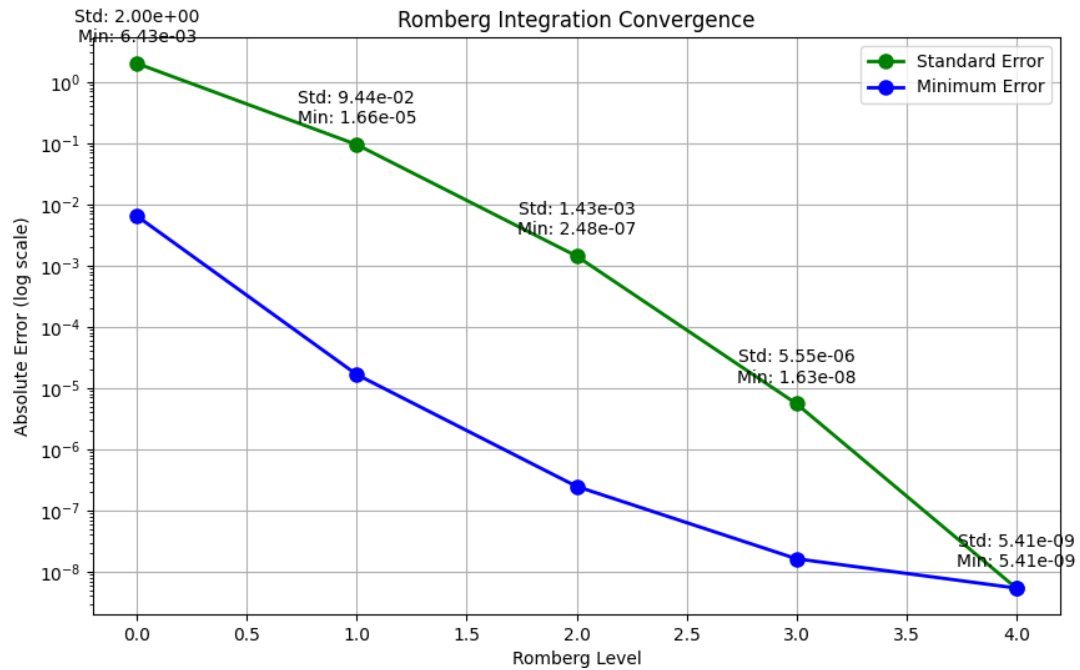
$f(x) = \sin(x)$

Romberg Integration Table

Exact Value: 2.0000000000

n	R(i,0)	R(i,1)	R(i,2)	R(i,3)	R(i,4)
n=1	0.0000000000	2.0943951024	1.9985707318	2.0000055500	1.9999999946
n=2	1.5707963268	2.0045597550	1.9999831309	2.0000000163	0.0000000000
n=4	1.8961188979	2.0002691699	1.9999997525	0.0000000000	0.0000000000
n=8	1.9742316019	2.0000165910	0.0000000000	0.0000000000	0.0000000000
n=16	1.9935703438	0.0000000000	0.0000000000	0.0000000000	0.0000000000



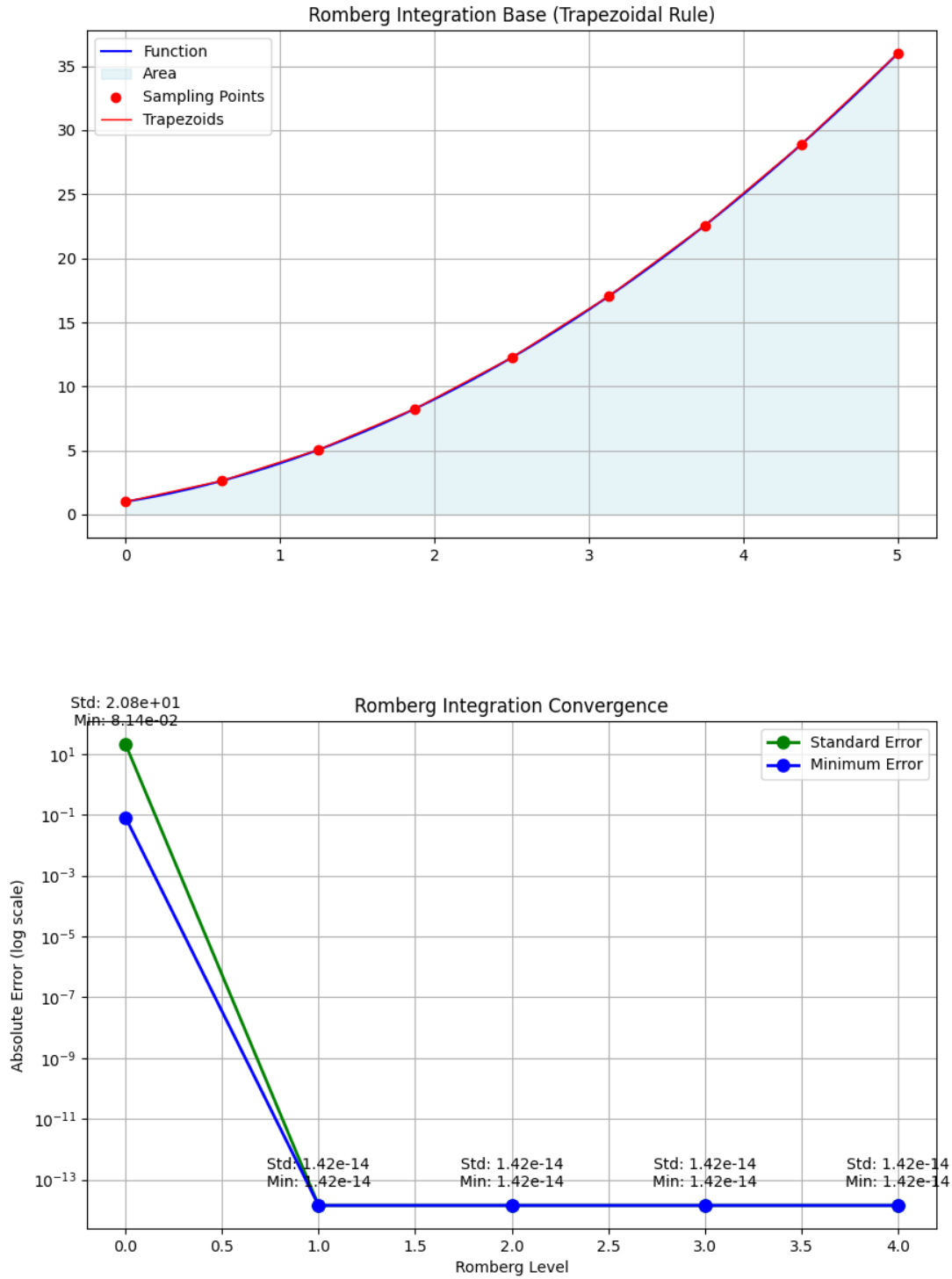


$$f(x) = x^2 + 2x + 1$$

Romberg Integration Table

Exact Value: 71.666666667

n	R(i,0)	R(i,1)	R(i,2)	R(i,3)	R(i,4)
n=1	92.5000000000	71.6666666667	71.6666666667	71.6666666667	71.6666666667
n=2	76.8750000000	71.6666666667	71.6666666667	71.6666666667	0.0000000000
n=4	72.9687500000	71.6666666667	71.6666666667	0.0000000000	0.0000000000
n=8	71.9921875000	71.6666666667	0.0000000000	0.0000000000	0.0000000000
n=16	71.7480468750	0.0000000000	0.0000000000	0.0000000000	0.0000000000

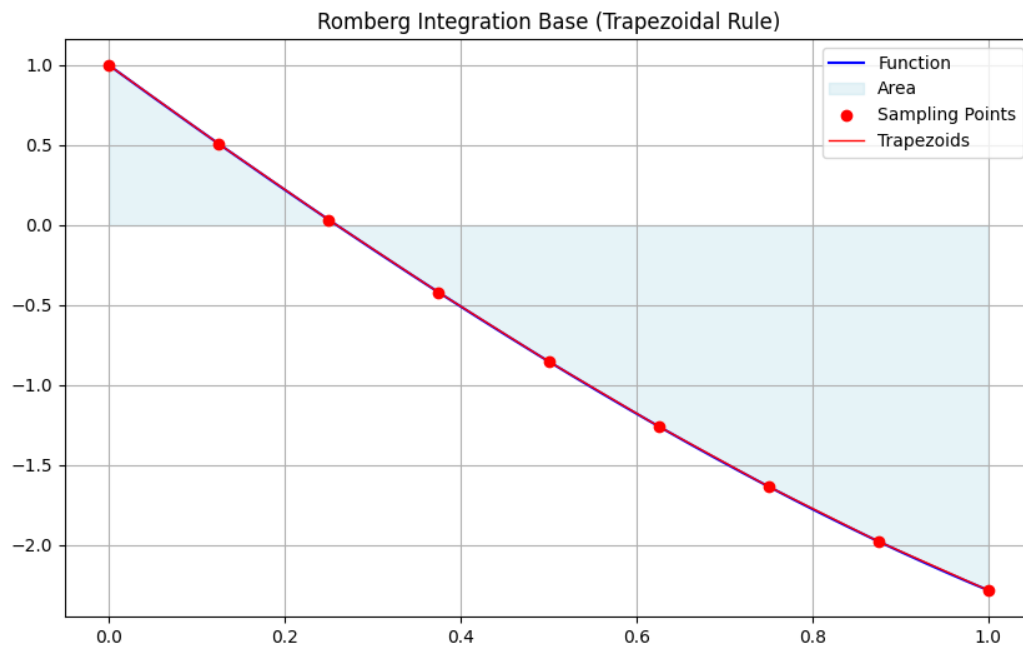


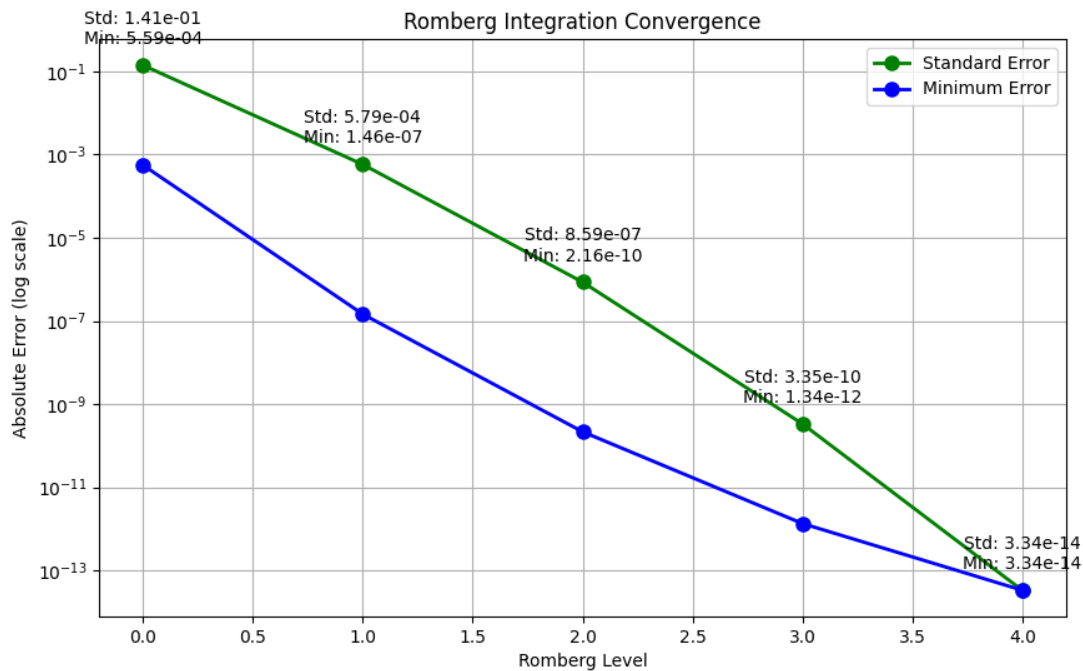
$$f(x) = e^x - 5x$$

Romberg Integration Table

Exact Value: -0.7817181715

n	R(i,0)	R(i,1)	R(i,2)	R(i,3)	R(i,4)
n=1	-0.6408590858	-0.7811388481	-0.7817173121	-0.7817181712	-0.7817181715
n=2	-0.7460689075	-0.7816811581	-0.7817181578	-0.7817181715	0.0000000000
n=4	-0.7727780954	-0.7817158453	-0.7817181713	0.0000000000	0.0000000000
n=8	-0.7794814078	-0.7817180259	0.0000000000	0.0000000000	0.0000000000
n=16	-0.7811588714	0.0000000000	0.0000000000	0.0000000000	0.0000000000





12.4 Interpretation of Results

Romberg integration is a powerful adaptive method for numerical integration that enhances the accuracy of the trapezoidal rule through **Richardson extrapolation**. It refines initial estimates using previously calculated values in the Romberg table, significantly reducing errors with each refinement level. This method provides high precision with relatively few computational steps.

The **Romberg table** consists of successive trapezoidal estimates that are combined to obtain progressively better approximations. The method is particularly efficient at reducing errors, often converging very quickly to the true integral value.

12.4.1 $f(x)=\sin(x)$

- **Exact Value:** 2.0000000000

Observation:

- The error significantly decreases with each refinement level, quickly converging to the exact value. By $n = 4$, the approximation becomes extremely close to the exact result, and by $n = 8$, the result is effectively exact.

12.4.2 $f(x) = x^2 + 2x + 1$

- **Exact Value:** 71.6666666667

Observation:

- Similar to the $\sin(x)$ case, the error decreases rapidly with each refinement. By $n = 2$, the result is already very close to the exact value, and after several additional refinements, the error is essentially eliminated.

12.4.3 $f(x) = e^x - 5x$

- **Exact Value:** -0.7817181715

Observation:

- The Romberg integration method reduces the error substantially with each level of refinement. By $n = 2$, the result is already very close, and by $n = 4$, the estimate is almost exact.

Conclusion

- **Romberg Integration** offers **rapid convergence** with each refinement level.
- The method refines the trapezoidal approximation through **Richardson extrapolation**, reducing the error exponentially as more levels are added.
- The error for all functions quickly approaches **machine precision**, making this method extremely efficient for accurate definite integral approximations.

13 Gaussian Elimination and Variants

13.1 Standard Gaussian Elimination

Gaussian Elimination is a numerical method for solving systems of linear equations of the form $Ax = b$. It transforms the coefficient matrix A into an upper triangular matrix using row operations (forward elimination), and then uses back substitution to solve for the unknowns in vector x .

This implementation performs Gaussian Elimination without pivoting, meaning it does not swap rows even when the pivot element is zero or close to zero. While this simplifies the algorithm, it may lead to numerical instability or division by zero for certain matrices.

The procedure consists of two main phases:

13.1.1 Forward Elimination

The goal of this phase is to eliminate the elements below the main diagonal by converting the matrix A into an upper triangular form.

For each pivot row k , the algorithm performs the following operations for all rows $i > k$:

- Compute the multiplier (also called the elimination factor):

$$\text{factor} = \frac{A[i][k]}{A[k][k]}$$

- Subtract the appropriate multiple of the pivot row from the current row:

$$A[i, k :] = A[i, k :] - factor \times A[k, k :]$$

$$b[i] = b[i] - factor \times b[k]$$

These steps zero out the sub-diagonal elements in column k, maintaining the equivalence of the system.

13.1.2 Back Substitution:

Once the matrix is in upper triangular form, the unknowns are determined starting from the last equation and substituting known values into the previous ones.

For each row i from n-1 down to 0:

$$x[i] = \frac{1}{A[i][i]} (b[i] - \sum_{j=i+1}^{n-1} A[i][j] \times x[j])$$

This method is best suited for well-conditioned matrices where pivoting is not essential. It serves as a foundational approach for understanding more robust numerical solvers.

Output:

Example 1	Example 2	Example 3
<pre>[5] A1 = np.array([[2, 3, 1], [4, 7, 5], [6, 18, 10]]) b1 = np.array([1, 3, 6]) x1 = gaussian_elimination(A1, b1) print("A:", A1) print("\nB:", b1) print("\nSolution:", x1)</pre>	<pre>[6] A2 = np.array([[10, 2, -1], [-3, -6, 2], [1, 1, 5]]) b2 = np.array([27, -61.5, -21.5]) x2 = gaussian_elimination(A2, b2) print("A:", A2) print("\nB:", b2) print("\nSolution:", x2)</pre>	<pre>[] A3 = np.array([[1, 1, 1], [0, 2, 5], [0, 0, 3]]) b3 = np.array([6, -4, 3]) x3 = gaussian_elimination(A3, b3) print("A:", A3) print("\nB:", b3) print("\nSolution:", x3)</pre>
<pre>A: [[2 3 1] [4 7 5] [6 18 10]] B: [1 3 6] Solution: [0.2 0.1 0.3]</pre>	<pre>A: [[10 2 -1] [-3 -6 2] [1 1 5]] B: [27. -61.5 -21.5] Solution: [0.5 8. -6.]</pre>	<pre>A: [[1 1 1] [0 2 5] [0 0 3]] B: [6 -4 3] Solution: [9.5 -4.5 1.]</pre>

14 Pivoting Strategies

14.1 Gaussian Elimination with Partial Pivoting:

Gaussian Elimination with Partial Pivoting is a numerical method used to solve systems of linear equations of the form $Ax = b$. The method extends basic Gaussian elimination by incorporating partial pivoting, a process that improves numerical stability by reducing the risk of division by small pivot elements. This is achieved by selecting the largest (in absolute value) element in the current pivot column and swapping rows

before proceeding with elimination. The method comprises two main phases: forward elimination with pivoting and back substitution.

14.1.1 Inputs:

- A: An $n \times n$ coefficient matrix.
- b: An $n \times 1$ right-hand side vector.

Both inputs are converted to floating-point arrays to ensure precision during division operations.

14.1.2 Forward Elimination with Partial Pivoting:

In each iteration of the elimination process (column-wise from top to bottom), the following steps are performed:

14.1.3 Partial Pivoting:

- Identify the row with the maximum absolute value in the current column (starting from the current pivot row to the bottom).

$$\text{max_row} = \text{argmax } |A[i][k]|$$

- Swap the current pivot row k with max_row in both the matrix A and the vector b . This reduces numerical errors and avoids division by small pivot elements.

14.1.4 Elimination:

- For each row below the pivot row, compute the elimination factor:

$$\text{factor} = \frac{A[i][k]}{A[k][k]}$$

- Use this factor to eliminate the current column entry:

$$A[i, k:] = A[i, k:] - \text{factor} \times A[k, k:]$$

$$b[i] = b[i] - \text{factor} \times b[k]$$

This transforms the matrix into an upper triangular form.

14.1.5 Back Substitution:

Once the matrix is upper triangular, solve for the unknowns starting from the last row:

$$x[i] = \frac{1}{A[i][i]} (b[i] - \sum_{j=i+1}^{n-1} A[i][j] \times x[j])$$

This recursive approach calculates each element of the solution vector x , moving from bottom to top.

14.1.6 Output:

The function returns the solution vector x , which satisfies the original system of equations $Ax = b$.

The image shows three examples of the `gaussian_elimination_partial_pivoting` function being used in Jupyter Notebooks. Each example displays the input matrices A and b , the function call, and the resulting solution vector x .

- Example 1:**

```
A1 = np.array([[2, 3, 1],
               [4, 7, 5],
               [6, 18, 10]])
b1 = np.array([1, 3, 6])
x1 = gaussian_elimination_partial_pivoting(A1, b1)

print("A =\n", A1)
print("\nb =\n", b1)
print("\nSolution x =\n", x1)
print()
```

A =
[[2 3 1]
 [4 7 5]
 [6 18 10]]

b = [1 3 6]

Solution x = [0.2 0.1 0.3]
- Example 2:**

```
A2 = np.array([[18, 2, -1],
               [-3, -6, 2],
               [3, 1, 5]])
b2 = np.array([27, -61.5, -21.5])
x2 = gaussian_elimination_partial_pivoting(A2, b2)

print("A:", A2)
print("\nb:", b2)
print("\nSolution:", x2)
```

A: [[18 2 -1]
 [-3 -6 2]
 [3 1 5]]

B: [27. -61.5 -21.5]

Solution: [0.5 8. -6.]
- Example 3:**

```
A3 = np.array([[1, 1, 1],
               [0, 2, 5],
               [0, 0, 3]])
b3 = np.array([6, -4, 3])
x3 = gaussian_elimination_partial_pivoting(A3, b3)

print("A:", A3)
print("\nb:", b3)
print("\nSolution:", x3)
```

A: [[1 1 1]
 [0 2 5]
 [0 0 3]]

B: [6 -4 3]

Solution: [0.5 -4.5 1.]

14.1.7 Advantages:

- Improves numerical stability over no pivoting.
- Avoids division by zero or small pivots.
- More efficient than complete pivoting.
- Works well for most practical problems.
- Reduces rounding errors significantly.

14.2 Gaussian Elimination with Complete Pivoting:

Gaussian Elimination is a fundamental method in numerical linear algebra for solving systems of linear equations. However, its numerical stability heavily depends on how pivot elements are chosen during the elimination process. Complete Pivoting is a pivoting strategy that enhances stability by selecting the largest (in magnitude) element in the remaining submatrix as the pivot, and rearranging both rows and columns accordingly.

The implemented method solves a linear system $Ax = b$ using Gaussian Elimination with Complete Pivoting, which improves numerical accuracy compared to partial or no pivoting.

14.2.1 Initialization:

- The matrix A and vector b are converted to float for numerical precision.
- A permutation vector P is initialized to keep track of column swaps.

14.2.2 Complete Pivoting Process:

- For each step k in the elimination:
 - A submatrix from the k -th row and column to the end is extracted.
 - The maximum absolute value in this submatrix is found to serve as the pivot.
 - Rows and columns are swapped so the pivot becomes the element at position $A[k, k]$.
 - The permutation vector P is updated to reflect column swaps.

14.2.3 Forward Elimination:

- For all rows below the current pivot row, a multiplier (factor) is calculated.
- This multiplier is used to eliminate the elements below the pivot, updating both A and b .

14.2.4 Back Substitution:

- Once A is in upper triangular form, the system is solved from the last row upward using standard back substitution.

14.2.5 Solution Reordering:

- Due to column swaps during pivoting, the solution vector x is reordered using the permutation vector P to match the original system's variable ordering.

14.2.6 Output:

```
Example 1
A1 = np.array([[2, 3, 1],
               [4, 7, 5],
               [6, 18, 10]])
b1 = np.array([1, 3, 6])
x1 = gaussian_elimination_complete_pivoting(A1, b1)
print("A:", A1)
print("\nb:", b1)
print("\nSolution:", x1)

A: [[ 2  3  1]
     [ 4  7  5]
     [ 6 18 10]]
b: [ 1  3  6]
Solution: [0.2 0.1 0.3]

Example 2
A2 = np.array([[18, 2, -1],
               [-3, -6, 2],
               [1, 1, 5]])
b2 = np.array([27, -61.5, -21.5])
x2 = gaussian_elimination_complete_pivoting(A2, b2)
print("A:", A2)
print("\nb:", b2)
print("\nSolution:", x2)

A: [[18  2 -1]
     [-3 -6  2]
     [ 1  1  5]]
b: [ 27.  -61.5 -21.5]
Solution: [ 0.5  8.  -6. ]

Example 3
A3 = np.array([[1, 1, 1],
               [0, 2, 5],
               [0, 0, 3]])
b3 = np.array([6, -4, 3])
x3 = gaussian_elimination_complete_pivoting(A3, b3)
print("A:", A3)
print("\nb:", b3)
print("\nSolution:", x3)

A: [[1 1 1]
     [0 2 5]
     [0 0 3]]
b: [ 6 -4  3]
Solution: [ 9.5 -4.5  1. ]
```

14.2.7 Advantages:

- Ensures maximum numerical stability.
- Minimizes rounding errors by selecting the largest pivot.
- Reduces risk of division by very small numbers.
- Provides accurate results for ill-conditioned matrices.

14.3 Gaussian Elimination with Scaled Partial Pivoting

Gaussian Elimination is a widely-used technique to solve systems of linear equations. To improve its numerical stability, Scaled Partial Pivoting is used as a pivoting strategy. In this method, the pivot selection is based on the scaled values of the rows, where each element is divided by the maximum absolute value in its row. This scaling reduces the impact of large differences in row magnitudes, ensuring that the algorithm avoids ill-conditioned matrices and improves stability in the elimination process.

The method `gaussian_elimination_scaled_partial_pivoting` solves a system of linear equations $Ax = b$ using Gaussian Elimination with the Scaled Partial Pivoting strategy, which improves numerical stability when solving systems with large differences in row magnitudes.

14.3.1 Initialization:

- The matrix A and vector b are cast to float to prevent precision issues during calculations.

14.3.2 Scaling:

- A scaling factor is computed for each row by taking the maximum absolute value from each row of matrix A . This scaling factor is used to normalize the elements in each row, ensuring that large coefficients don't dominate the pivot selection.

14.3.3 Pivot Selection with Partial Pivoting:

- For each step k , the ratio of the absolute value of the element $A[i, k]$ to the scaling factor for row i is computed for rows below and including row k .
- The row with the highest ratio is selected as the pivot row, ensuring that the largest value relative to the row's scale is used as the pivot.
- If the selected pivot row is different from the current row k , the rows are swapped, and the scaling factor for the swapped rows is also updated.

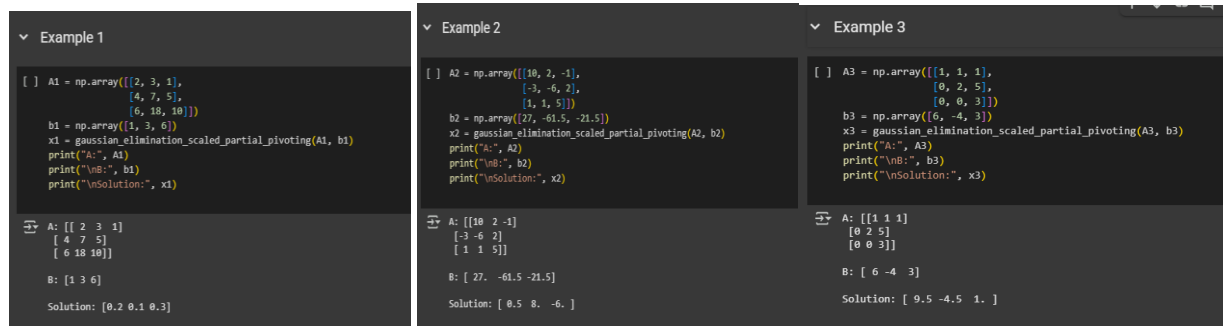
14.3.4 Forward Elimination:

- The elements below the pivot in the current column are eliminated by subtracting an appropriate multiple of the pivot row from each of the lower rows.
- The right-hand side vector b is updated correspondingly.

14.3.5 Back Substitution:

- After the matrix A has been transformed into upper triangular form, the system is solved using back substitution. Starting from the last row, the unknowns are computed in reverse order by solving for each variable.

14.3.6 Output:



```
Example 1
[ ] A1 = np.array([[2, 3, 1],
                  [4, 7, 5],
                  [6, 18, 10]])
b1 = np.array([1, 3, 6])
x1 = gaussian_elimination_scaled_partial_pivoting(A1, b1)
print("A:", A1)
print("b:", b1)
print("Solution:", x1)

A: [[ 2  3  1]
     [ 4  7  5]
     [ 6 18 10]]
b: [ 1  3  6]
Solution: [ 0.2  0.1  0.3]

Example 2
[ ] A2 = np.array([[10, 2, -1],
                  [-3, -6, 2],
                  [1, 1, 5]])
b2 = np.array([27, -61.5, -21.5])
x2 = gaussian_elimination_scaled_partial_pivoting(A2, b2)
print("A:", A2)
print("b:", b2)
print("Solution:", x2)

A: [[10  2 -1]
     [-3 -6  2]
     [ 1  1  5]]
b: [ 27.  -61.5 -21.5]
Solution: [ 0.5  8.  -6. ]

Example 3
[ ] A3 = np.array([[1, 1, 1],
                  [0, 2, 5],
                  [0, 0, 3]])
b3 = np.array([6, -4, 3])
x3 = gaussian_elimination_scaled_partial_pivoting(A3, b3)
print("A:", A3)
print("b:", b3)
print("Solution:", x3)

A: [[ 1  1  1]
     [ 0  2  5]
     [ 0  0  3]]
b: [ 6 -4  3]
Solution: [ 9.5 -4.5  1. ]
```

14.3.7 Advantages:

- Further enhances numerical stability over basic partial pivoting.
- Normalizes row values to avoid bias from large entries.
- Reduces errors in ill-scaled or unbalanced systems.
- Minimizes the impact of scaling differences between rows.
- More robust for solving real-world systems with varied magnitudes.

15 Matrix Factorization

15.1 LU Decomposition

LU Decomposition is a matrix factorization method that decomposes a given square matrix A into the product of two matrices: a lower triangular matrix L and an upper triangular matrix U . This decomposition is particularly useful for solving systems of linear equations, matrix inversion, and computing determinants. The LU Decomposition is performed without requiring pivoting (assuming the matrix is invertible), which makes it computationally efficient.

The method `lu_decomposition` computes the LU decomposition of a square matrix A , where the matrix A is expressed as the product of a lower triangular matrix L and an upper triangular matrix U . This is a crucial technique for solving linear systems efficiently.

15.1.1 Initialization:

- The matrix A is assumed to be square ($n \times n$).
- Two matrices L and U are initialized as $n \times n$ matrices filled with zeros. Matrix L will hold the lower triangular matrix, and matrix U will hold the upper triangular matrix.

15.1.2 Iterative Decomposition:

- **Upper Triangular Matrix (U):** For each row i , the elements in the i -th row of U are computed by subtracting the product of the corresponding elements of L and U from the elements of A . This process continues until the upper triangle of the matrix is filled.

- For each k-th element in the i-th row of U, the sum of previously computed elements in L[i, j] and U[j, k] is subtracted from A[i, k] to obtain the element U[i, k].
- Lower Triangular Matrix (L):** For each element below the diagonal of L, the values are computed by subtracting the product of L and U elements from A, then dividing by the pivot element U[i, i] (the diagonal element of U).
 - For each element in L[k, i], the value is calculated as $(A[k, i] - \text{sum}) / U[i, i]$, where sum is the sum of previously computed elements in L[k, j] and U[j, i].

15.1.3 Diagonal of L:

The diagonal elements of the matrix L are always set to 1, as required by the definition of a lower triangular matrix in LU decomposition.

15.1.4 Return Matrices:

- The function returns the two matrices L and U, which together represent the LU decomposition of matrix A.

15.1.5 Matrix Printing:

The print_matrix function is provided to neatly format and display the matrices L and U. It formats each matrix element to two decimal places for better readability, printing the matrices with their respective names for clarity.

Output:

The image shows three examples of LU decomposition in a Jupyter Notebook environment. Each example consists of a code cell and a corresponding output cell.

Example 1:

```
A1 = [
    [2, -1, -2],
    [-4, 6, 3],
    [-4, -2, 8]
]
L1, U1 = lu_decomposition(A1)
print_matrix(L1, "L1")
print_matrix(U1, "U1")
```

L1:

```
[ ' 1.00', ' 0.00', ' 0.00' ]
[ ' -2.00', ' 1.00', ' 0.00' ]
[ ' -2.00', ' -1.00', ' 1.00' ]
```

U1:

```
[ ' 2.00', ' -1.00', ' -2.00' ]
[ ' 0.00', ' 4.00', ' -1.00' ]
[ ' 0.00', ' 0.00', ' 3.00' ]
```

Example 2:

```
A2 = [
    [10, -1, 2],
    [-1, 11, -1],
    [2, -1, 10]
]
L2, U2 = lu_decomposition(A2)
print_matrix(L2, "L2")
print_matrix(U2, "U2")
```

L2:

```
[ ' 1.00', ' 0.00', ' 0.00' ]
[ ' -0.10', ' 1.00', ' 0.00' ]
[ ' 0.20', ' -0.07', ' 1.00' ]
```

U2:

```
[ ' 10.00', ' -1.00', ' 2.00' ]
[ ' 0.00', ' 10.90', ' -0.80' ]
[ ' 0.00', ' 0.00', ' 9.54' ]
```

Example 3:

```
A3 = [
    [1, 1/2, 1/3],
    [1/2, 1/3, 1/4],
    [1/3, 1/4, 1/5]
]
L3, U3 = lu_decomposition(A3)
print_matrix(L3, "L3")
print_matrix(U3, "U3")
```

L3:

```
[ ' 1.00', ' 0.00', ' 0.00' ]
[ ' 0.50', ' 1.00', ' 0.00' ]
[ ' 0.33', ' 1.00', ' 1.00' ]
```

U3:

```
[ ' 1.00', ' 0.50', ' 0.33' ]
[ ' 0.00', ' 0.08', ' 0.08' ]
[ ' 0.00', ' 0.00', ' 0.01' ]
```

15.1.6 Advantages:

- LU decomposition is a fast and efficient method for solving systems of linear equations, particularly when multiple systems need to be solved with the same coefficient matrix A.
- This method is often used in numerical algorithms and provides a direct way to solve for the variables in $Ax = b$ by using forward and back substitution.

15.2 DoLittle Decomposition

The Doolittle LU Decomposition is a specific form of LU decomposition where a square matrix A is factorized into two matrices: a lower triangular matrix L with unit diagonal elements (1's on the diagonal) and an upper triangular matrix U . This method is efficient for solving systems of linear equations, inverting matrices, and computing determinants.

The Doolittle method builds L and U such that:

$$A = LU$$

with $L[i][i] = 1$ for all i . The decomposition proceeds row by row for U and column by column for L , ensuring that the lower triangle is filled below the main diagonal and the upper triangle is filled including the diagonal.

15.2.1 Initialization:

- Two matrices L and U of size $n \times n$ are initialized with zeros.
- The diagonal entries of L are explicitly set to 1 ($L[i][i] = 1.0$), as per Doolittle's formulation

15.2.2 Construction of U Matrix:

- For each row i , compute the upper triangular matrix $U[i][j]$ for all columns $j \geq i$.
- Each element is calculated by subtracting the sum of previously computed terms from the corresponding element in A :

$$U[i][j] = A[i][j] - \sum_{k=0}^{i-1} L[i][k] \times U[k][j]$$

15.2.3 Construction of L Matrix:

- For each column i , compute the lower triangular matrix $L[j][i]$ for rows $j > i$.
- Each element is computed as:

$$L[j][i] = \frac{A[j][i] - \sum_{k=0}^{i-1} L[j][k] \times U[k][i]}{U[i][i]}$$

- A ZeroDivisionError is raised if $U[i][i] = 0$, as it indicates that the matrix is singular or needs pivoting.

15.2.4 Return Values:

- The function returns the matrices L and U such that $A = LU$.

15.2.5 Output:

Example 1

```
A1 = [
    [2, -1, -2],
    [-4, 6, 3],
    [-4, -2, 8]
]
L1, U1 = doolittle_lu_decomposition(A1)
print_matrix(L1, "L1")
print_matrix(U1, "U1")
```

L1:

```
[[' 1.00', ' 0.00', ' 0.00'],
 [' -2.00', ' 1.00', ' 0.00'],
 [' -2.00', ' -1.00', ' 1.00']]
```

U1:

```
[[' 2.00', ' -1.00', ' -2.00'],
 [' 0.00', ' 4.00', ' -1.00'],
 [' 0.00', ' 0.00', ' 3.00']]
```

Example 2

```
A2 = [
    [10, -1, 2],
    [-1, 11, -1],
    [2, -1, 10]
]
L2, U2 = doolittle_lu_decomposition(A2)
print_matrix(L2, "L2")
print_matrix(U2, "U2")
```

L2:

```
[[' 1.00', ' 0.00', ' 0.00'],
 [' -0.10', ' 1.00', ' 0.00'],
 [' 0.20', ' -0.07', ' 1.00']]
```

U2:

```
[[' 10.00', ' -1.00', ' 2.00'],
 [' 0.00', ' 10.90', ' -0.80'],
 [' 0.00', ' 0.00', ' 9.54']]
```

Example 3

```
A3 = [
    [1, 1/2, 1/3],
    [1/2, 1/3, 1/4],
    [1/3, 1/4, 1/5]
]
L3, U3 = doolittle_lu_decomposition(A3)
print_matrix(L3, "L3")
print_matrix(U3, "U3")
```

L3:

```
[[' 1.00', ' 0.00', ' 0.00'],
 [' 0.50', ' 1.00', ' 0.00'],
 [' 0.33', ' 1.00', ' 1.00']]
```

U3:

```
[[' 1.00', ' 0.50', ' 0.33'],
 [' 0.00', ' 0.08', ' 0.08'],
 [' 0.00', ' 0.00', ' 0.01']]
```

15.2.6 Advantages:

- Provides a systematic approach for solving linear systems $Ax = b$ using two triangular solves: first $Ly = b$ (forward substitution) and then $Ux = y$ (back substitution).
- Efficient for repeated solving when A is constant and multiple right-hand side vectors b are used.

15.3 Crout LU Decomposition

Crout's LU Decomposition is a matrix factorization technique in which a square matrix A is decomposed into the product of a lower triangular matrix L and an upper triangular matrix U, where the upper triangular matrix has unit diagonal entries (i.e., all diagonal elements of U are 1). Unlike Doolittle's method, Crout's algorithm fills L column-wise and computes U row-wise.

The decomposition satisfies the equation:

$$A = LU$$

with $U[j][j] = 1$ for all j.

15.3.1 Initialization:

- Matrices L and U of size $n \times n$ are initialized with zeros.
- The diagonal entries of U are set to 1 ($U[j][j] = 1.0$) as required by Crout's formulation.

15.3.2 Compute the Lower Triangular Matrix L:

- For each column j, compute each entry of L from row j to n using:

$$L[i][j] = A[i][j] - \sum_{k=0}^{j-1} L[i][k] \times U[k][j]$$

- This step populates L column-wise.

15.3.3 Compute the Upper Triangular Matrix U:

- For each column j, and rows $i > j$, compute:

$$U[i][j] = \frac{A[j][i] - \sum_{k=0}^{j-1} L[j][k] \times U[k][i]}{L[j][j]}$$

- A check is performed to ensure the pivot element $L[j][j] \neq 0$, otherwise a `ZeroDivisionError` is raised.

15.3.4 Return Values:

- The function returns matrices L and U such that $A = LU$.

15.3.5 Output:

Example 1

```
[ ] A1 = [
    [2, -1, -2],
    [-4, 6, 3],
    [-4, -2, 8]
]
L1, U1 = crout_decomposition(A1)
print_matrix(L1, "L1")
print_matrix(U1, "U1")
```

L1:
[' 2.00', ' 0.00', ' 0.00'
[' -4.00', ' 6.00', ' 0.00'
[' -4.00', ' -2.00', ' 8.00'

U1:
[' 1.00', ' 0.00', ' 0.00'
[' -0.50', ' 1.00', ' 0.00'
[' -1.00', ' 0.50', ' 1.00'

Example 2

```
[ ] A2 = [
    [10, -1, 2],
    [-1, 11, -1],
    [2, -1, 10]
]
L2, U2 = crout_decomposition(A2)
print_matrix(L2, "L2")
print_matrix(U2, "U2")
```

L2:
[' 10.00', ' 0.00', ' 0.00'
[' -1.00', ' 11.00', ' 0.00'
[' 2.00', ' -1.00', ' 10.00'

U2:
[' 1.00', ' 0.00', ' 0.00'
[' -0.10', ' 1.00', ' 0.00'
[' 0.20', ' -0.09', ' 1.00'

Example 3

```
[ ] A3 = [
    [1, 1/2, 1/3],
    [1/2, 1/3, 1/4],
    [1/3, 1/4, 1/5]
]
L3, U3 = crout_decomposition(A3)
print_matrix(L3, "L3")
print_matrix(U3, "U3")
```

L3:
[' 1.00', ' 0.00', ' 0.00'
[' 0.50', ' 0.33', ' 0.00'
[' 0.33', ' 0.25', ' 0.20'

U3:
[' 1.00', ' 0.00', ' 0.00'
[' 0.50', ' 1.00', ' 0.00'
[' 0.33', ' 0.75', ' 1.00'

15.3.6 Advantages:

- Useful for solving systems of linear equations, especially when the decomposition is reused for multiple right-hand sides.
- More numerically stable than Doolittle's method for certain matrices due to the way the pivot elements are used in L.

15.4 Cholesky LU Decomposition

Cholesky decomposition is a numerical method for decomposing a symmetric, positive-definite matrix AAA into the product of a lower triangular matrix LLL and its transpose:

$$A = LL^T$$

This method is especially efficient for solving systems of linear equations, matrix inversion, and numerical simulations involving symmetric positive-definite matrices.

15.4.1 Initialize:

- A zero matrix L of the same dimension as A is created to store the result.

15.4.2 Iterative Computation:

- For each row index i and column index $j \leq i$, compute the value of $L[i][j]$ using the Cholesky formula.
- **Diagonal Elements** ($i == j$):

$$L[i][j] = \sqrt{A[i][i] - \sum_{k=0}^{j-1} L[i][k]^2}$$

This step ensures the diagonal values of L are real and positive, which is necessary for the matrix to be positive-definite. If the computed value under the square root is zero or negative, the function raises a ValueError.

- **Off-Diagonal Elements** ($i > j$):

$$L[i][j] = \frac{A[i][j] - \sum_{k=0}^{j-1} L[i][k] \times L[j][k]}{L[j][j]}$$

This computes the remaining lower triangular values using previously computed entries.

15.4.3 Return:

The function returns the matrix L. The transpose L^T can be obtained easily if needed for further operations.

Output:

Example 1	Example 2	Example 3
<pre>[] A1 = [[25, 15, -5], [15, 18, 0], [-5, 0, 11]]</pre>	<pre>[] A2 = [[6, 15, 55], [15, 55, 225], [55, 225, 979]]</pre>	<pre>[] A3 = [[1, 1/2, 1/3], [1/2, 1/3, 1/4], [1/3, 1/4, 1/5]]</pre>
<pre>L1 = cholesky_decomposition(A1) print_matrix(L1, "L1")</pre>	<pre>L2 = cholesky_decomposition(A2) print_matrix(L2, "L2")</pre>	<pre>L3 = cholesky_decomposition(A3) print_matrix(L3, "L3")</pre>
<pre>L1: [' 5.00', ' 0.00', ' 0.00'] [' 3.00', ' 3.00', ' 0.00'] [' -1.00', ' 1.00', ' 3.00']</pre>	<pre>L2: [' 2.45', ' 0.00', ' 0.00'] [' 6.12', ' 4.18', ' 0.00'] [' 22.45', ' 20.92', ' 6.11']</pre>	<pre>L3: [' 1.00', ' 0.00', ' 0.00'] [' 0.50', ' 0.29', ' 0.00'] [' 0.33', ' 0.29', ' 0.07']</pre>

15.4.4 Advantages:

- Faster and more memory-efficient than LU decomposition for symmetric, positive-definite matrices.
- Requires about half the computations compared to standard LU decomposition.
- Numerically stable due to reduced rounding error accumulation.